Using PDDL models to solve TUSS

How to model TUSS as an Automated Planning problem and solve it

Nazariy Lonyuk

Using PDDL models to solve TUSS

How to model TUSS as an Automated Planning problem and solve it

by

Nazariy Lonyuk

Nazariy Lonyuk 4596811

Instructor: M. de Weerdt
Teaching Assistant: I. Hanou
Project Duration: April, 2024 - December, 2024
Faculty: Faculty of Electrical Engineering, Maths and Computer Science, Delft

TU Delft

Preface

With the completion of this thesis, I have fulfilled the final requirement to obtain my master's degree in Embedded Systems at Delft University of Technology. In this thesis, I explore the modelling and solving of Train Unit Shunting and Servicing problem instances using open-source planners, as well as a planner I developed myself.

First and foremost, I would like to thank my supervisors, Mathijs de Weerdt and Issa Hanou. Before I began this project, I had never heard of train unit shunting, and they were instrumental in helping me familiarise myself with the subject matter. I thoroughly enjoyed the many discussions we had about this rich topic.

Most importantly, I would like to thank them for keeping me grounded and focused on what was important to complete this project, something I greatly needed. Their guidance and support have significantly elevated the quality of this work to a level I could not have achieved on my own.

I hope that reading this thesis proves to be both an enjoyable and educational experience.

Nazariy Lonyuk
Delft, December 2024

Summary

Due to the increased demand for train travel, train operators are considering increasing their rolling stock. Before achieving this, they must enhance the capacity of their shunting yards. This is attempted by improving methodologies for solving the Train Unit Shunting and Servicing (TUSS) problem. To address the TUSS problem, a planner determines routes on shunting yards for trains, ensuring they visit designated service tracks before parking in a configuration that facilitates a smooth departure.

TUSS is a well-studied problem, and various approaches have been proposed. The first approach capable of solving real-world, complete TUSS instances is a local search method introduced by van den Broek et al. [54]. In this thesis, we explore an alternative approach using PDDL models. PDDL is the standard language for describing Automated Planning problems. Automated Planning is a well-established field within artificial intelligence, and new, improved algorithms are continually developed to solve PDDL models for problems similar to TUSS.

In this thesis, we design a detailed model in PDDL and propose several methods to simplify the model so that planning algorithms perform more efficiently compared to the detailed model. When solving simplified models, a post-processing routine is employed to generate detailed shunting plans. The performance of several model-independent PDDL planners was analysed, and the best-performing planner was identified as Temporal FastDownward [41].

By analysing plans obtained from experiments, we identified areas for improvement. Based on this knowledge, we developed a new TUSS-specific planner called Train Order Preserving Search (TOPS). TOPS employs a search algorithm with effective pruning of symmetrical states and a custom heuristic that guides the search towards states where the order of trains aligns with the departure order. TOPS significantly outperformed Temporal FastDownward in these experiments.

Contents

1

Introduction

In the Netherlands, the largest passenger railway operator is Nederlandse Spoorwegen (NS), which transports over a million passengers every day [50]. In the morning and afternoon, most of their trains are travelling across the many tracks on the Dutch network. At night, however, the demand is low, and the non-operational trains need to be parked on so-called shunting yards. While on the yard, a significant portion of the trains need to be cleaned or repaired before they can depart the next morning to transport passengers.

The railway network is currently very busy, but before NS can consider increasing their rolling stock, they must ensure that any additional trains can be parked and serviced during non-operational hours. However, increasing the capacity of a shunting yard is easier said than done. Shunting yards are often built in populated areas near busy train stations, which often makes physically increasing their size extremely costly or outright impossible. Nonetheless, there is an alternative way to increase shunting yard capacity: improving computational methods for planning.

For many shunting yards, the capacity is not limited by the space on the tracks but rather by the ability to compute shunting plans. A shunting plan consists of a time-stamped list of actions to be performed. Devising such a plan is difficult because the planner must consider many requirements. All trains requiring service must visit a designated service track, execution of the movements specified in a shunting plan must not cause collisions, and all actions must be completed before the trains are scheduled to depart the yard. Designing a plan that moves trains into a valid parking configuration, performs all service tasks, and ensures no collisions is referred to as solving the Train Unit Shunting and Servicing (TUSS) problem.

TUSS is a well-studied NP-hard feasibility problem [23] for which no exact solver currently exists. The current state of the art is a solving method based on a local-search algorithm proposed by van den Broek et al. [54]. Other approaches in the literature include Multi-Agent Pathfinding (MAPF) algorithms [40] or Mixed-Integer Linear Programming (MILP) solvers [2], among others. Despite the effort invested in TUSS research, there is still room for improvement. Many approaches are impractical because they are specific to one shunting yard, perform poorly on real-world instances, or do not guarantee feasible solutions.

What has not been tried before, however, is framing TUSS as an automated planning problem. In a planning problem, a planner must find a sequence of actions that one or more agents can execute to achieve a specific goal within an environment, given initial conditions and constraints. Automated planning, or AI planning, is a widely studied field of artificial ntelligence that focuses on developing algorithms to solve planning problems that are too complex for human planners. New and improved planning algorithms are actively developed to this day. International Planning Competitions (IPCs) are regularly organised to compare planning algorithms on a variety of problems. Most state-of-the-art planning algorithms are model-independent, which means that they are able to solve problems using only their descriptions.

The standard language used to describe planning problems is the Planning Domain Definition Language (PDDL) [1]. Over the years, many PDDL models have been created and used that are similar to TUSS in some aspects. For instance, many problems featured in IPCs require planners to determine

the optimal path for one or more agents to reach a destination while also visiting specific intermediate locations. This resembles how, in a TUSS problem, trains must visit a service track before finding a location to park.
The active development and versatility of planning algorithms, as well as the existence of moderately similar PDDL models, indicate great potential for solving TUSS problem instances. For this reason, this thesis explores how TUSS problems can be solved using PDDL models and AI planning algorithms. To the best of our knowledge, planning algorithms have never been applied to solve TUSS problems. Before we can apply PDDL planners we consider our first research question:

RQ 1: How can we model TUSS problems using PDDL?

We take a careful look at the aspects we can include to model a TUSS problem. PDDL is a rich language that allows for detailed problem descriptions. However, detailed problem descriptions can limit scalability for solving complex instances as search times can increase exponentially. Therefore also explore ways to simplify descriptions of certain aspects of TUSS. We consider different variations of models to answer the following research question:

RQ 2: How can a TUSS model in PDDL be simplified to improve planner performance?

The simplified models are used to produce partial-order shunting plans for TUSS problem instances. In this thesis we also describe a post-processing routine based on a Constraint Programming (CP) model that is able to finalize these partial-order plans into full-order solutions.
After exploring the modelling of TUSS in PDDL, we conduct experiments to answer the following questions:

RQ 3: Which PDDL planners and model variations perform best when solving TUSS problem instances?

Performance of different planner and model combinations will be measured based on the complexity of problems they can solve within a set time limit and the quality of the solutions. We compare PDDL planners as well as the impact of the previously considered simplifications. After analysis of the experimental results we provide a recommendation for the best planner and model that can be used to solve TUSS problems.
Additionally, the contents of the partial-order plans are examined to identify properties of TUSS problems that negatively impact algorithm performance in the experiments. This analysis is used to answer the last research question:

RQ 4: Can we use the knowledge obtained from analysing solutions by model-independent planners to create a better performing TUSS-specific planner?

We develop the Train Order Preserving Search (TOPS) planner to specifically solve the TUSS we have created using PDDL. This planner will use a modified version of the A^* search algorithm and a custom heuristic evaluation to solve TUSS problem instances. This heuristic evaluates states such that it prefers to obtain or maintain the departure order of trains in the problem instance.
By answering the aforementioned research questions this paper makes the following contributions:

1. Extensive exploration of detailed modelling of TUSS problems using PDDL
2. A large set of TUSS models with varying degrees of detail written in PDDL
3. A new methodology for solving TUSS problem instances using any compatible AI planner and a CP model
4. Extensive analysis of performance per planner and per model variation
5. A new TUSS-specific planner able to solve problem instances modelled with PDDL

The remainder of this thesis is structured as follows: In Chapter 2, we explain important concepts related to automated planning, PDDL, and CP in more detail. Since many AI planners, including TOPS, use A^* search, we also explain how this algorithm is applied in a planning context in the same chapter. In the next chapter, Chapter 3, we present an overview of relevant literature. Because many variations of TUSS have been studied, Chapter 4 provides a detailed problem description of the specific variation used in this thesis. In Chapter 5 we explore how we can model TUSS in PDDL and describe the post-processing routine. The experimental setup and analysis of results are presented in Chapter 6. The

algorithm and heuristic evaluation used in TOPS are explained in Chapter 7 as well as an analysis of its performance compared to the best performing model-independent planners. Finally, Chapter 8 presents our conclusions and recommendations for future research.

2 Background

Throughout this thesis we discuss modelling automated planning problems in the Planning Domain Definition Language (PDDL). These models are solved by planners that often use the A^* search algorithm. In our solving method we also include a Constraint Programming (CP) model. This chapter is intended to provide the reader with foundational knowledge on the key concepts regarding these topics. After reading the sections below we believe the reader to have a sufficient understanding of the technical details required to follow the next chapters.

2.1. Automated planning

Planning was described by Ghallab [21] as the reasoning side of acting, a deliberate process in which actions are organised into a plan in order to best achieve some desired objectives. A **plan** is described by Tate [53] as a representation of future behaviour, usually as a set of actions with temporal or other constraints on them to be executed by one or multiple agents.
Human planners such as those tasked with computing train unit shunting plans face many complexities when planning due to a large amount of actors and demanding requirements. Fortunately advances in information processing technology gives them access to affordable and efficient planning tools. This was one of the main motivations which set off a large amount of research in the field of automated planning. In **automated planning** the aim is to develop artificially intelligent algorithms to perform planning tasks.

2.1.1. Actions

An **action** is described by Ghallab et al. [22] as something an agent does that makes a change in the environment and its own state. An **agent** is any entity that can interact with its own environment. In our TUSS models this can be a train or a driver for example.
Information about certain properties of agents and the environment are stored in a **state**. Actions change properties of agents or the environment such that a new state is reached. In other words, actions are seen as **state transitions**.
When defining an action we describe the acting agents, preconditions and effects. The **preconditions** describe in what states the action can be performed. The **effects** describe how the state changes after the action has been performed.

2.1.2. Domains

In a **domain** the environment, rules and capabilities of a planning problem are described. The environment is described by a set of **literals**, which have boolean values, and **fluents**, which have numeric values.
Furthermore, a domain contains action definitions. For each action definition the required values for specific literals and fluents are listed in the **preconditions**. If a state does not meet these preconditions, the action can not be executed. How a state changes after the execution of an action is listed in its definition under the **effects**.

A domain which includes any numeric fluent is labelled as **numeric**. If the action definitions in a domain include its duration, when the preconditions must hold and when effects take place we call this a **temporal domain**. A domain which is neither numeric nor temporal is called a **classical domain**.

2.1.3. Problem instance

A planning problem is completely described by a domain and a problem instance. We can view the domain as the rule book for the environment and the **problem instance** as a description of initial environment and the objective. In it is included the initial state and a description of the goal.
The **goal** is composed of several goal literals. **Goal literals** can either be literals as named in the domain descriptions or a numeric comparison between the values of two fluents. A state for which all goal literals have the truth value is called a **goal state**.
One thing worth to note is that the same label applies to a domain and the corresponding problem instance. A problem instance which follows the state-transition system in a temporal domain is considered a **temporal problem**. The same applies to the **numeric** and **classical** labels.

2.1.4. Objective

A plan can be any sequence of actions, but not every plan is a solution. A plan can be called a **solution** if a goal state is reached after performing the sequence of actions starting out from the initial state.
If the objective is to find a feasible plan we are happy if any solution is found. Problems which describe such an objective are called **satisficing problems**. However a problem can also be formulated as an **optimization problem** which means that the objective is to find an optimal solution. An **optimal solution** is one where the cost is minimized. The cost for numeric problems is often a literal defined in the domain which increases or decreases in value in the effects of actions. For temporal domains this metric is often the total duration.

2.2. PDDL

the Planning Domain Definition Language (PDDL) was first introduced in 1998 by Ghallab et al. [1] in an attempt to standardise formulation of automated planning problems. Since then several new versions have been released with many additional features. In this section we will discuss the elements necessary to describe a planning problem in PDDL as well as several optional features and their implications. To define a planning problem in PDDL we need two files: a domain and a problem file.

2.2.1. Domain file

The **domain** file can be seen as the rule book of the environment. A domain must include all of the following:

- List of requirements
- List of object types
- List of literals
- action definitions

The list of **requirements** is preceded by the $:requirements$ tag and is used to indicate to the planner which PDDL features are used in the domain definition. In practice however, many planners ignore the requirements list and infer the used features from the domain definition.
PDDL objects are used as parameters in literals, fluents and actions. These objects can be assigned a type so that in the definitions for literals, fluents and actions it can be detailed which objects are allowed to participate in them and which are not.
The **literals** are listed under the $:predicates$ tag. Each literal has a name and can have one, multiple or no parameters. Parameters are easily recognised as they are prefixed with a question mark.
Action definitions are listed independently, each of them preceded with an $:action$ tag. Below this tag we define a list of typed or untyped parameters with the $:parameters$ tag. An action's preconditions are listed under the $:preconditions$ tag its effects under the $:effects$ tag.
The syntax for temporal domains differs slightly regarding action definitions. Instead of the $:action$ tag we use the $:durative\text{-}action$ tag. After the parameters are listed a numerical value must be assigned for the $:duration$. The list of preconditions are listed under the $:conditions$ tag. For each precondition

we must indicate either that it must hold before ($at-start$), during ($over-all$) or after ($at-end$) executing the action. Usage of the :$effects$ tag is similar to regular actions but for each it must be indicated whether it takes place before or after execution.
Optionally a domain file may also contain a list of fluents or derived literals. **Fluents** are listed under the :$functions$ tag and similarly to literals they can have one, multiple or no parameters. The difference between fluents and literals is that fluents are assigned a numeric value. A numeric comparison between fluents can be one of the preconditions listed in an action definition. Before including fluents in a domain the :$numeric$-$fluents$ requirement must be included in the list of requirements.
Finally, a domain might include definitions for **derived literals**. For derived literals we also define a set of parameters much like for literals and fluents. Similarly to action definitions every derived literal is defined separately and is preceded by the :$derived$ tag. The first line includes the name of the derived literal and its parameters. Below that we include all literals or numeric comparisons that must hold for the derived literal to hold. The truth value of a derived literal is updated after any state transition. When including derived literals the domain we must include the :$derived$-$predicates$ tag in the list of requirements.

2.2.2. Problem file

The **problem** file contains the following:

- List of objects
- Description of the initial state
- Description of what makes a goal state

In the **list of objects**, indicated by the :$objects$ tag, we assign a name and a type to every object participating in the problem. In PDDL it is not possible to dynamically create or delete objects so this list must contain all objects that will ever be relevant for the problem.
After the objects have been defined we must provide a description of the **initial state**, which is preceded by the :$init$ tag. In the initial state we include all grounded literals which are true in the initial state. Any grounded literal not included in the initial state is assumed to be false.
To describe a **grounded** literal we write the name of the literal followed by the name(s) of objects for which this literal hold. This is different from how literals are defined in the domain, where they are followed by (a) parameter(s) which are not directly linked to a specific named object.
Besides grounded literals the initial state also contains assignments of numbers to all grounded numeric fluents. Numeric comparisons cannot be written in the initial state.
However they can be written in the **goal description** alongside literals or their negations. Any state in which all the mentioned literals and comparisons hold is a valid **goal state**.
One last feature which may be included in the problem file is a metric. A metric is recognized by the :$metric$ tag and followed by instructions for the planner to either $minimize$ or $maximize$ a numeric expression. This numeric expression can either be a single fluent or a more complex expression like the sum of multiple fluents for example. If this feature is included in the problem file we have described the problem as an optimization problem. Without a metric the problem file always describes a satisficing problem.

2.3. A* search

Originally proposed by Hart et al. [24], A^* search is an algorithm meant to search both mathematically and heuristically for the minimum cost path in a weighted graph. Finding an optimal path in a graph is the goal for many categories of engineering problems. One such category is Multi-Agent PathFinding (MAPF), which is described by Ma [38] as planning collision-free paths for multiple robots. TUSS can be seen as a real-world application of this problem as is done by Mulderij et al. [40].
In this section we provide several definitions related to graphs before explaining the algorithm. Lastly we will look at the concepts related to weighted graphs in a planning and MAPF context to explore the role of A^* search in solving TUSS.

2.3.1. Weighted graphs

A graph is described by Hart et al. [24] as a set of elements $\{n_i\}$ called **nodes** and a set of directed line segments $\{e_{ij}\}$ called **arcs**. Arc e_{ij} connects node n_i to n_j. In weighted graphs every arc is assigned a cost c_{ij}.

2.3.2. Algorithm

Pseudo-code

Below we provide pseudo-code of the A^* search algorithm based on the description by Hart et al. [24]:

Algorithm 1 A^* search

```
1: given s, T
2: Initialize OPEN, CLOSED, PATH
3: OPEN ← s
4: PATH ← s
5: while OPEN not empty do
6:     n_current ← node with lowest priority in OPEN
7:     for n_i in successors of n_current do
8:         if n_i in T then
9:             PATH ← n_i
10:            terminate and return PATH
11:        end if
12:        if n_i not in OPEN and not in CLOSED then
13:            calculate f(n_i)
14:        end if
15:        priority(n_i) ← f(n_i)
16:        OPEN ← n_i
17:    end for
18:    CLOSED ← n_current
19:    OPEN ← OPEN \ {n_current}
20: end while
```

- **Step 1:** In the problem definition we are provided with start node s and the set of terminal nodes T
- **Step 2-4:** We initialize a **PATH** which at first contains only the start node s. During search we keep track of nodes which are candidates to be expanded in the **OPEN** list. We also have a **CLOSED** list in which we keep track of the nodes which have already been explored.
- **Step 5:** Search continues until we have explored all nodes
- **Step 6:** The node with lowest priority is placed in front of the queue and thus selected first for expansion. The value of the priority is set based on the outcome of the evaluation function.
- **Step 7:** Any node that can be reached from the current node is a successor of the current node.
- **Step 8-11:** Once we have reached a goal state we return the solution. In its original form A^* search terminates when the first solution is found although in many practical applications the algorithm is extended such that search continues.
- **Step 12-14:** If we have not encountered the successor node before we calculate the evaluation function, otherwise we look it up.
- **Step 15-17:** Evaluate the successor node and place it in its correct place in the queue.
- **Step 18-19:** After we have evaluated all possible continuations starting from the current node, we remove it from the queue as it has already been fully explored. We do not want to redo the evaluation of the current node and thus place it in the $CLOSED$ list.

Evaluation function

The area of the search space to which search is directed depends mostly on the calculated evaluations. Let us delve deeper into the definition of the evaluation function $f(n_i)$:

$$f(n_i) = g(n_i) + h(n_i) \tag{2.1}$$

Function $g(n_i)$ can be seen as the mathematical part of the evaluation function. Its value is the cost it takes to reach node n_i from a previously defined start node s. If we define a path P as $\{n_0, n_1, ..., n_N\}$ we can write the cost as follows:

$$g(n_i) = \sum_{j=1}^{N} c_{(j-1)j} \tag{2.2}$$

One thing to note is that $n_0 = s$ and $n_N = n_i$.
The heuristic part of the evaluation function is represented by $h(n_i)$ and there are many ways in which it can be defined. When defining a $h(n_i)$ the aim is to estimate the cost it takes to reach any node in the set of defined terminal nodes T when starting out from node n_i.

2.3.3. Planning problem as a weighted graph

Before we can use algorithms designed for path-finding in graphs we must define what is considered a **node** and what is an **arc** in a planning context.
A **node** in a planning graph consists of both a **state** and the sequence of **actions** that were performed to reach it. It is common that states in a planning problem can be reached via multiple paths. To be able to distinguish two states that were reached via different action sequences we store the sequences in the node.
After performing an **actions** in a specific state we end up in a new state and the sequence leading up to that state is also different. In other words, we transitioned from one node to another. In a planning context we see actions as **arcs**. The **weight** of an arc is defined as the cost that is induced when performing an action.
Now we can see how a planning problem can be seen as a path-finding problem for graphs when we switch perspectives. Finding the shortest path from a starting node to another can be seen as finding the optimal sequence of actions to reach a goal state when starting from an initial state.

2.3.4. Evaluating heuristics

There is no one definition for $h(n_i)$ as there is for $g(n_i)$. Heuristics are specific to one planner and sometimes even to only to a single planning problem.
What we want to achieve with a search algorithm is to find the shortest path as quickly as possible. Heuristics that can accurately evaluate a node allow for the search algorithm to make more informed decisions. When successors of nodes not included in this path are evaluated we are wasting time. For this reason the performance of a heuristic is often measured by the amount of node expansions [26]. We call a heuristic **good** if it leads us to expanding few nodes to find the shortest path. Determining whether a heuristic is good or not can be done experimentally.
Besides not wanting detours to be explored we want the optimal path to be included in the search space that is explored by the algorithm. A heuristic for which it can be guaranteed that the optimal solution is found is called **admissible**. Proof is provided by Hart et al. [24] that any heuristic which never overestimates the cost is admissible.
An admissible heuristic is not necessarily good. To give an example, a heuristic which estimates zero cost for every node is admissible but does not allow for informed decision making during search. Moreover, a good heuristic does not necessarily mean that performance will be good when using an unmodified A^* search. Helmert et al. [26] for example show how performance can worsen exponentially even with almost perfect heuristics.

2.4. Constraint Programming

Constraint Programming (CP) is described by Rossi et al. [44] as a powerful paradigm for solving combinatorial search problems using a wide variety of artificial intelligence techniques. The concept behind CP is that the user creates a model, specifies variables and constraints, and uses a general-purpose solver to optimise the value of a variable. CP models are applied to solve many different types

of problems; one example is resource-constrained scheduling.

In this section, the contents of a CP model are explored in more detail, along with several modelling techniques that improve the performance of a solver.

2.4.1. Model Contents

In a CP model, the user defines variables, each of which has a numeric value. For each variable, a domain is specified, describing the possible values that the variable can take.

In addition, a model includes constraints that specify rules determining which combinations of values for different variables are allowed or disallowed.

Finally, a CP model can provide instructions on how the solver should perform its search for a solution.

2.4.2. Modelling Techniques

Several techniques enhance the performance of a CP model. One such technique is the inclusion of redundant constraints. These are constraints that are already implied by others but when explicitly included it might make information available to the solver earlier, improving performance.

Another effective technique is the inclusion of symmetry-breaking constraints. In problems involving identical resources, many symmetrical solutions exist that are equivalent in quality. By adding these constraints, the solver can focus its search more effectively, leading to improved performance.

3

Literature review

Before we analyse the performance of several planning algorithms on TUSS problem instances, we present an overview of other approaches that have been taken. This allows us to better understand the context in which to place the research done in this thesis.

The Train Unit Shunting Problem (TUSP) was, to the best of our knowledge, first introduced by Freling et al. [19]. In this paper they define and solve the matching and parking sub-problems. To solve the matching sub-problem they describe how train units in incoming train configurations are organised in outgoing train configurations. They solve the parking sub-problem by assigning where each incoming train unit will be parked until departure. For both sub-problems mathematical model is composed and solved with a Mixed Integer Programming (MIP) solver. This approach does not consider much detail and does not guarantee the feasibility of the resulting shunting plans.
Lentink et al. [35] extend the TUSP description by including the routing sub-problem. They take a sequential approach where they first compute solutions to the matching and parking sub-problems. In every step the solution to the previous sub-problem is used as a starting point for the next. In the last step a local search strategy is used where routes over the shunting yard are computed using the Occupied Network A^* search algorithm. This methodology provided more support to human planners compared to the one proposed by Freling et al. [19], but it may not be able to find routes for all trains defined in the problem instance
The first solving method which computes solutions to the parking and matching sub-problems simultaneously was proposed by Kroon et al. [31]. However, this integrated approach required a large amount of constraints to be generated. This meant that for realistic scenarios it took too long to find solutions.
Because of the scalability issues with MIP-based solving methods, Jacobsen et al. [29] explored an approach where a local search algorithm is guided by several metaheuristics. They find that their approach produces solutions of slightly lower quality compared to a MIP-solver but significantly faster. For further research they suggest integrating crew planning when solving TUSP instances.
The experiments performed by Haahr et al. [23] also show that their heuristic approach outperforms exact solvers regarding computational time. In their paper they compare a greedy construction heuristic, a Constraint Programming (CP) formulation and a column generation approach. They suggest further research to include crew planning and scheduling of servicing tasks.
The current state of the art is a fully integrated local search approach proposed by van den Broek et al. [54]. They partly build upon the suggestion made by Haahr et al. by including the servicing sub-problem. This sub-problem is solved if all trains mentioned in a provided service schedule have been serviced between arrival and departure. Transposing the TUSP to the Train Unit Shunting and Servicing (TUSS) problem adds to the complexity of the complete problem since the servicing of trains can only be done on certain designated service tracks. The local search algorithm performs quite well on realistic scenarios for shunting yards in the Netherlands. In their recommendations for further research they acknowledge that actions performed by crew have a significant impact in a shunting plan.
Several other techniques have been proposed since, such as the Multi-Agent PathFinding (MAPF) approach by Cuilenborg [14] using Conflict-Based Search (CBS). Another example is an Evolutionary

Algorithm (EA) approach with Conflict-Based Crossover (CBC) by Athmer [4]. However both solving methods did not improve on the performance of the current state of the art.

Several of the authors have suggested integrating crew planning when solving TUSP or TUSS instances. One approach was proposed by van den Broek et al. [11]. In their solving method they propose assigning crew members to train activities with heuristic search methods.
Szabó [51] explores a local search approach to schedule maintenance crew whereas van Nes [42] incorporates the branch-and-price algorithm to schedule tasks for train drivers.

4

Problem definition

The Train Unit Shunting and Servicing (TUSS) problem is a well-studied problem, and numerous research papers address its solutions. As became clear in the literature review, many different versions of it have been studied with varying levels of detail. In this chapter we provide a detailed and formal definition of the problem as it is be studied in the rest of this paper.
Firstly, the problem setting is discussed. This is followed by a specification of the provided input and required output.
A common approach in TUSS involves decomposing the problem into smaller sub-problems, solving them independently or in pairs. This methodology is also relevant in our approach and thus the chapter concludes with a presentation of the sub-problems into which TUSS is most commonly divided.

4.1. Setting

In TUSS, the objective is to fulfil specific requirements by moving trains across a shunting yard. This section explores the components of a shunting yard and the types of trains considered in subsequent chapters of this paper.

4.1.1. Shunting yard

A shunting yard is a type of railroad yard where trains go during non-operational hours. It comprises multiple track components of various types. In our models, the following types are considered:

- Track
- Switch
- English switch

A **track** is mostly straight, has a fixed length, and is connected to one other component on either side. Trains cannot be side-by-side on the same track; hence, the sum of the lengths of trains located on a track cannot exceed its length. A description of a shunting yard must specify whether parking and servicing tasks are allowed on each track. For every shunting yard, there is one **entry track**, which is used for trains entering and exiting the yard.
A **switch** is a mechanical installation connected to one track component on one side and to two track components on the other, though not simultaneously. Parking is not permitted on switches. This paper considers only electrified switches, which can be controlled remotely, as these are the most common type in Dutch shunting yards.
An **English switch** connects to two track components on either side. Like a regular switch, it enables connections between one track component and another at a time, and parking is not permitted on it. Only electrified English switches, remotely controlled, are considered in this paper.
A visual representation of both types of switches is shown in figure 4.1. The upper image shows a regular switch where depending on its position either tracks $1a$ and $3a$ are connected or $1a$ and $4a$. For the English switch in the lower image only one of $1b$ - $3b$, $1b$ - $4b$, $2b$ - $3b$ or $2b$ - $4b$ is possible at one

point in time. It is important to note that sharp turns are not possible. Using the example presented in figure 4.1: to get from track $3a$ to $4a$ a train must first move onto $1a$.

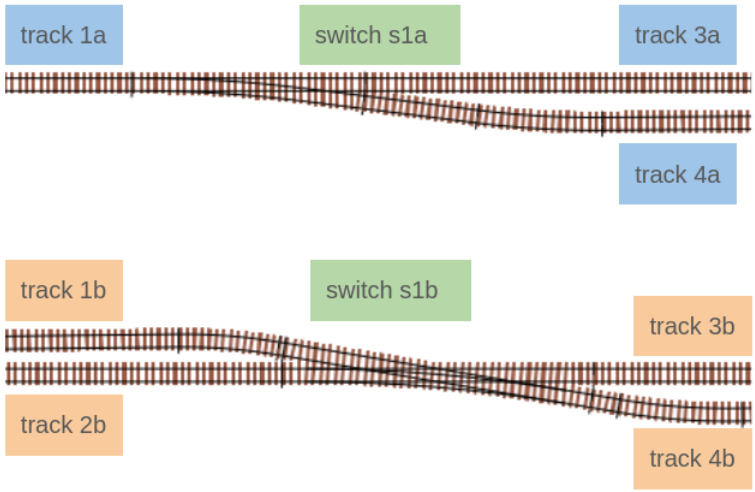

Figure 4.1: Two types of switches that can be found on a shunting yard: regular switch (upper image) and an English switch (lower image)

4.2. Trains

A **train composition**, often referred to simply as a **train**, consists of one or multiple connected **train units**. A train can be decomposed into multiple trains by breaking the connection between two train units, a process known as **uncoupling**. Conversely, connecting trains to form a single composition is referred to as **coupling**.
A **train unit** has a specific type, and all train units of the same type share the same length. Train units cannot change type or be decomposed into smaller components. This paper focuses exclusively on self-propelling electric train units, meaning any unit can move independently without requiring a specific other unit in its composition.
However, before a train can move it must be operated by a **train driver**. Self-driving trains are not included in any of our TUSS descriptions.

4.3. Input

Now that we have defined the important components of TUSS we continue with a formal description of the input which is provided to a (human or AI) planner before solving a problem instance:

- Arrival schedule
- Departure schedule
- Servicing task list
- Shunting yard layout

The **arrival schedule** specifies train compositions and their arrival times at the entry track. Train compositions detail the specific train units they contain and their configuration.
The **departure schedule** describes the required train compositions and departure times from the yard. Unlike the arrival schedule, the departure schedule specifies required types and configurations rather than individual train units. Multiple configurations may satisfy the departure requirements. This paper assumes that the first train departs only after the last train has arrived, a condition referred to as the *midnight condition* by Di Stefano et al. [15].
The **servicing task list** outlines the servicing tasks required for each train unit. It specifies the types of services needed, as it is uncommon for all service types to be available on every servicing track. Examples of service types include maintenance and internal or external cleaning.

A **shunting yard layout** is often provided as a list of track descriptions. The relevant properties detailed in the track description are:

- Track name
- Track type
- Track length
- Whether parking is possible
- List of service types available
- Possible track connections

Often most of the regular tracks in a shunting yard can be parked on whereas (English) switches can never be parked on. Servicing tracks are commonly very few when compared to tracks where no servicing tasks can be performed.

4.4. Output

Given the previously described input a planner is tasked with providing a sequence of shunting activities that ensures the provided schedules can be followed under the relevant constraints. The shunting activities a planner can use in their plan are:

- A driver entering- or exiting a train
- A driver walking from one train to another
- Moving a train
- Parking a train
- Servicing a train
- (Un)coupling a train to create new compositions
- Moving a switch to change track connections

4.5. Sub-problems

As inferred from the input and output descriptions, there are numerous aspects and interactions to consider when solving a TUSS instance. These complexities make TUSS a challenging problem, often decomposed into a set of sub-problems that are solved independently before combining all sub-solutions to form a complete solution.
The Train Unit Shunting Problem was introduced by Freling et al. [19], who formulated the parking and matching sub-problems. Lentink et al. [35] later extended this formulation to include the routing and servicing sub-problems.

4.5.1. Matching

All arriving train units have a name and a type. In the departure schedule, train compositions are described as a sequence of train unit types. A solution to the matching problem assigns a position in a departing train composition to each incoming train unit. An example arrival schedule is shown in Table 4.1. A solution for the departure schedule in Table 4.2 is provided in Table 4.3.

Table 4.1: Simple arrival schedule containing two different compositions.

Arrival Time	Train	Train unit types
18:00	(1,2)	(ICM-3, ICM-3)
18:15	(3)	(ICM-4)

Table 4.2: Simple departure schedule containing two different compositions.

Departure Time	Train	Train unit types
09:00	A	(ICM-4, ICM-3)
09:15	B	(ICM-3)

Table 4.3: Solution to the matching problem described by the arrival schedule in Table 4.1 and the departure schedule in Table 4.2.

Train unit	Train	Position
1	A	2
2	B	1
3	A	1

4.5.2. Parking

In the parking sub-problem, the aim is to assign a parking track and position on it to every incoming train unit. Several constraints disallow certain parking assignments, as detailed by Lentink [34].
First, some tracks in a shunting yard cannot be parked on, such as switches. Secondly, train units have a fixed length and cannot overlap; therefore, the sum of train unit lengths on a parking track can never exceed the length of the track. Lastly, it must not be possible that a train cannot arrive at or depart from the shunting yard because it is blocked by another parked train.
Freling et al. [19] address the matching sub-problem first, whereas later approaches, such as that of Kroon et al. [32], often solve the matching and parking sub-problems simultaneously.

4.5.3. Routing

When solving the routing sub-problem, the objective is to find a path for every train that ensures it reaches its destinations on time and without collisions.
A collision occurs when two trains occupy the same piece of track at the same time. In practice, when a train driver intends to move a train, they submit a route request. If accepted, the tracks included in this request are reserved exclusively for the train in question. Once the train reaches its destination, the tracks become available again. Lentink [34] provides further details on this process.

4.5.4. Servicing

A servicing schedule specifies the train units, the servicing tasks to be performed on them, and the locations in the shunting yard where these tasks can be carried out. A solution to the servicing sub-problem describes where and in what order each train unit is serviced.
Some approaches, such as that of Lentink [34], combine the routing and servicing sub-problems by treating a servicing track as an intermediate destination and a parking track as the final destination.

5

Modelling TUSS in PDDL

In this chapter, we explore the modeling of TUSS problem instances using PDDL. First, we attempt to develop a detailed model that accurately represents a TUSS instance, enabling PDDL planners to generate a shunting plan that is both feasible and provides solutions to all sub-problems.
Next, we examine variations of this detailed model, simplifying or omitting certain TUSS aspects in hopes of improving planner performance. However, we acknowledge that plans obtained using these simplified models are less detailed and lack the guarantees provided by the detailed model. Additionally, comparing plans from models with differing levels of detail is inherently unfair.
To address this, we incorporate a post-processing routine in our solving method. This routine treats PDDL plans as partial solutions and extends them to include the missing details. It also optimizes the action sequence using a Constraint Programming (CP) model. We detail the steps taken to modify PDDL plans, ensuring they achieve the same level of detail as those from the detailed model, allowing fair comparisons.

5.1. Detailed modelling of TUSS in PDDL

We aim to create a detailed TUSS model in PDDL to determine whether PDDL's provides the necessary tools to produce plans that solve all four sub-problems. A PDDL plan achieves this by:

- Solving the **parking** sub-problem by assigning parking locations that prevent the need for rearrangements before departure.
- Addressing the **servicing** sub-problem by ensuring specified trains visit a service track during the shunting period.
- Handling the **matching** sub-problem by detailing when, where, and how train units are uncoupled or combined to meet the departure schedule.
- Resolving the **routing** sub-problem by specifying paths for all trains such that no collisions occur.

It is only recently, for example by van den Broek et al. [11] or Szabo [51], that crew planning was integrated when solving TUSS instances. Our model also adopts an integrated approach, although only considering train drivers and service workers that stay on one track during shunting.
The most critical measure of plan quality is its feasibility. A shunting plan is feasible if all required shunting activities can be performed within the provided timeframe without collisions and within the allocated resources. When constructing the domain and problem files, we consider the following questions:

1. Is the duration correctly estimated?
2. Can actions be executed within the given problem description and resource constraints?
3. Are collisions avoided if the plan is followed?

The detailed PDDL model is deemed complete only if these questions can be affirmatively answered. In this section, we discuss the TUSS aspects essential to include in the PDDL model and the features required to represent them.

5.1.1. Concurrent Actions

Modeling concurrent actions is a practical necessity. A model without concurrency capabilities will avoid underestimating plan durations but will struggle to solve instances with many train units due to the disparity in durations between service tasks and other shunting activities. For instance, van den Broek et al. [54] report that service tasks can take up to 56 minutes, while movements can take 5 minutes or less. Efficient plans exploit this disparity by assigning drivers to move other trains while one is being serviced.

To enable concurrency, we include the `:durative-actions` requirement in the domain and define `:durative-actions` instead of `:actions`. As explained in Section 2.2.1, `:durative-actions` require specifying a duration, which can be a numeric constant or fluent. Additionally, we must define when `:preconditions` and `:effects` are relevant during the action. Correctly formulating these elements ensures that concurrency is modeled accurately without underestimating durations or introducing collision risks.

5.1.2. Train Locations

Accurately tracking train locations requires specifying the track and the position on that track. Using relative positions, such as the length of the track and a train's relative location, can lead to inaccuracies in plan durations. For instance, this method may indicate sufficient space on a track but fail to account for necessary train movements to prevent collisions. To avoid such discrepancies, we model the exact one-dimensional coordinates of trains.

In our model, we represent train locations with two numeric fluents: `train_length`, the train's length, and `aside_distance`, the distance from the A-side of the train to the A-side of the track. The B-side location can be derived from these fluents. This level of detail ensures no additional actions are needed to resolve train positioning, resulting in precise plans.

5.1.3. Track Capacity

Tracks cannot accommodate side-by-side trains; all trains on a track must be positioned sequentially. Therefore, the total length of all trains on a track must not exceed the track's length. Domain rules should enforce this, ensuring plans do not cause collisions.

Each track's length is stored as a numeric constant, `track_length`. We also maintain the contiguous positions of trains on a track using two numeric fluents: `astack_distance`, the A-side position of the train stack, and `bstack_distance`, the B-side position of the stack. These fluents dynamically update as trains move on or off a track, ensuring track capacity constraints are respected.

5.1.4. Train Movement Clearance

Executing a shunting plan dynamically alters the shunting yard's state. Movement actions must validate certain prerequisites:

- A driver must operate the train.
- The driver must be positioned at the train's front.
- No other train should obstruct the path.
- Sufficient space must exist on the destination track.

We track whether a train is operated using the literal `operated(train)`. Initially, no train is operated, but this state changes when a driver performs an `enter` action. Additionally, a train can only move in the direction indicated by the `direction_aside` or `direction_bside` literals.

To simplify planner decision-making, movements are restricted to track-to-track actions between connected tracks. These movements are only allowed if sufficient space exists on the destination track, verified using the `bstack_distance` fluent.

Listing 5.1 provides an example of movement predicates, while Figure 5.1 visually represents the fluents used for capacity and location tracking.

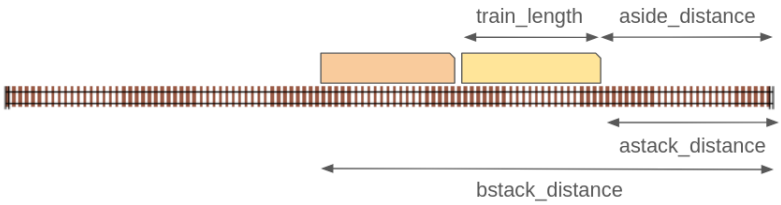

Figure 5.1: A visual representation of the fluents used to ensure exact train locations and track capacity is never exceeded.

Listing 5.1: Necessary predicates that verify whether a movement can happen or not.

```
(:durative-action move_aside
  :parameters (?t - train ?from ?to - track)
  ...
  :condition (and
    ...
    (over all (operated ?t))
    (over all (direction_aside ?t))
    (over all (connected ?from ?to))
    (over all (>= (track_length ?to) (train_length ?t)))
    (or
        (at start (>= (+ (train_length ?t)(bstack_distance ?to)) (track_length ?to)))
        (at start (= (num_trains ?to) 0))
    )
    (at start (= (aside_distance ?t)(astack_distance ?from)))
  )
  :effect (and
    ...
  )
)
```

5.2. Integration of Personnel Scheduling

In previous research, the scheduling of personnel is often not considered when producing a shunting plan, with job assignment solved as a separate problem afterward. Kamenga [2] mentions only maintenance crew, while the state-of-the-art local search algorithm by vd Broek et al. [54] ignores personnel planning entirely, though it notes that walking times of drivers can significantly impact shunting plans. Extensive research into personnel scheduling as a separate problem has been conducted by vd Broek et al. [11], Szabo [51], and Nes [42], among others. Despite differences in solving algorithms, these studies share similar requirements: considering which tasks each crew member can perform, the travel time between tasks, and ensuring no crew member is overworked.

In our detailed model, we integrate a simplified version of the personnel scheduling problem, excluding considerations of employee satisfaction. Additionally, we distinguish between two types of crew: **drivers** and **service workers**. Drivers are responsible for driving trains, coupling, and uncoupling them. When drivers need to move between locations on the shunting yard, they do so by walking. Service workers, on the other hand, remain at a designated platform near a service track, where they can enter trains to perform servicing tasks.

Several considerations regarding drivers are necessary to maintain realism. Drivers can interact with any train entering the yard and can be located at any point in the shunting yard. To keep track of drivers, we include `driver` objects, allowing us to assign literals to them.

First, the model must ensure that a driver cannot operate two trains simultaneously. To enforce this, we introduce a literal called `idle` for each `driver` object. This literal serves as a $:precondition$ for entering a train and is set to *false* once the driver has entered. When a driver exits a train, the model must identify which driver becomes available to operate other trains and which train is no longer operable. For this purpose, we define another literal, `driving`, for every possible `driver-train` object pair. This literal is set to *true* when the specific driver is inside the specific train.

Lastly, to include walking durations for drivers, the model must track the location of each driver. This is

accomplished using a single literal, `driver_at`, which has two parameters: a `driver` object and a `track` object. This literal is set to *true* if the driver is located on a given track. A driver's location becomes undefined when they enter a train and is reset when they exit. While it is theoretically possible to know the exact location of every driver, since train locations are precisely tracked, we choose to only track the driver's location at the level of the track. To prevent underestimation of walking times, these times are slightly overestimated.

As in the aforementioned research, walking times between every possible pair of tracks are precomputed and stored in numeric fluents called `walking_duration`, which take two $track$ objects as parameters. Unlike the approach proposed by Nes [42], we do not include the possibility of drivers tagging along with another train. Consequently, travel times for drivers may be slightly overestimated.

5.2.1. Simplified Constraints

To avoid overcomplicating the domain, our model does not optimize for employee satisfaction or include constraints regarding working hours and break times for the crew. We assume that the inclusion of multiple `driver` objects will facilitate plans that meet reasonable working hours, or that minor modifications to task assignments to satisfy such constraints will not significantly impact the total duration of the shunting plan.

A code snippet demonstrating how the mentioned literals are updated is provided in Listing 5.2. It was decided not to include service crew as separate objects in the PDDL domain. Unlike drivers, they stay at one platform in the shunting yard. Similar to drivers we must ensure that the predicates in the domain do not allow for one service crew to work on multiple trains at the same time. With drivers we had to solve this by introducing objects, but for service crews we can solve this by predicates involving only trains. Unlike drivers, service workers enter and exit trains only on one specific location: special designated service tracks. If we limit the number of trains that can be located on a service track at the same time we can guarantee that a service crew never is servicing two trains at the same time. This limit can be increased or decreased depending on the amount of service crews working on a service track.

5.2.2. Preventing crossings

A "crossing" occurs when a movement by one vehicle is obstructed by another, as described for buses by Gallo et al. [20]. In papers about the Train Unit Shunting System (TUSS), this is often referred to as a train collision. In previous sections, we have described the rules for train movement. These rules only allow trains to move from one track to another; trains can only move if there is no other train on the same track between it and its destination track, and trains can only move onto another track if there is enough space on the side they enter. Once on the destination track, they move up to the first train they find on it if the track is not empty. Collisions are impossible on any track included in the PDDL problem file because of the rules described in the domain file. A simple method of guaranteeing that no collisions occur at any point in a plan is to include every piece of infrastructure described in the shunting yard layout in the problem file.

However, this would significantly increase the number of objects in the problem instance and require many more decisions to be made by the planner. The Kleine Binckhorst shunting yard, for example, is described by 18 tracks and 49 switches. Switches are relatively small compared to tracks, and they are never an intermediate or final destination for a train since they cannot be parked or turned there. Therefore, another approach that does not include switches is preferred, but we must perform some additional steps to guarantee that no collisions occur during the time every train spends between tracks.

We introduce a literal called `potential_connection`, which, like the `connected` literal, takes two `track` objects as parameters. The `connected` literal describes an active connection over which trains can move, while this is not necessarily the case for a `potential_connection`, which describes connections that could be made at some point in time. Any track can only have one active connection, represented by the `connected` literal, on each side of it.

If we examine an example of some rail infrastructure in Figure 5.2, it becomes clear. The literal `potential_connection` is true for any combination of one (L)eft track and one (R)ight track. In order to guarantee that a plan generated using our domain does not contain any collisions, we must

Listing 5.2: Updating of literals such that crew can be planned realistically.

```
(:durative-action enter_aside
  :parameters (?d - driver ?t - train ?r - track)
  ...
  :condition (and
    ...
    (over all (train_at ?t ?r))
    (at start (driver_at ?d ?r))
    (at start (idle ?d))
  )
  :effect (and
    ...
    (at start (not (idle ?d)))
    (at start (not (driver_at ?d ?r)))
    (at end (operated ?t))
    (at end (direction_aside ?t))
    (at end (drivig ?d ?t))
  )
)
(:durative-action exit
  :parameters (?d - driver ?t - train ?r - track)
  ...
  :condition (and
    ...
    (at start (drivig ?d ?t))
    (over all (train_at ?t ?r))
  )
  :effect (and
    ...
    (at start (not (drivig ?d ?t)))
    (at end (driver_at ?d ?r))
    (at end (idle ?d))
  )
)
(:durative-action walk
  :parameters (?d - driver ?from ?to - track)
  :duration (= ?duration (walking_distance ?from ?to))
  :condition (and
    ...
    (at start (idle ?d))
    (at start (driver_at ?d ?from))
  )
  :effect (and
    ...
    (at start (not (idle ?d)))
    (at start (not (driver_at ?d ?from)))
    (at end (driver_at ?d ?to))
    (at end (not (idle ?d)))
  )
)
```

design predicates such that only one movement between any of the tracks on the left and any of the tracks on the right can happen concurrently.

To achieve this, ”knots” such as the one shown in Figure 5.2 are replaced by one imaginary track in the model. This imaginary track has special properties, as it can only contain at most one train regardless of that train’s length; trains cannot park or turn on it, and drivers cannot exit a train while it is on this track. How this looks visually can be seen in Figure 5.3. Now there are potential connections between tracks L1-3 and track M, as well as between track M and tracks R1 and R2.

To ensure a collision-free plan, we define the initial state such that the only active connections allowed are between L1-M and M-R1. In the domain, we include `switch` actions that can deactivate one active connection and replace it by making another potential connection active. This means that we have established the rule that at most one active connection can exist on each side of a track. This `switch` action cannot occur while a train is moving on either of the tracks. To ensure this, we set a literal called

`blocked` to be true for a track if a movement is happening.

When these definitions are included in the domain, we can guarantee that no collisions occur on any tracks or switches that are either included or excluded in the problem file. A code snippet for the relevant actions and literals can be seen in Listing 5.3.

Figure 5.2: A section of a shunting yard where a train could move from any of the track son the left to any of the tracks on the right.

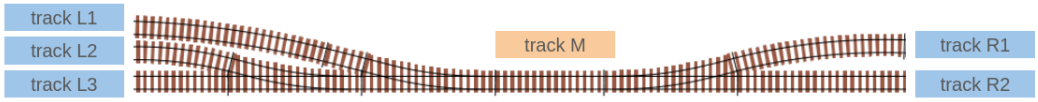

Figure 5.3: The same infrastructure as in figure 5.2 but now the knot has been replaced by a track

Listing 5.3: Updating of literals to ensure collision free movement across switches.

```
(:durative-action move
  :parameters (?t - train ?from ?to - track)
  ...
  :effect (and
    ...
    (at start (blocked ?from)) ; block all tracks involved for action duration
    ...
    (at end (not (blocked ?to)))
  )
)
(:durative-action switch
  :parameters (?a1 ?a2 ?b - track)
  ...
  :condition (and
    ...
    (over all (potential_connection ?a1 ?b))
    (over all (potential_connection ?a2 ?b))
    (over all (not (blocked ?a1))) ;assure none of the tracks involved are blocked at any
        point
    ...
    (at start (connected ?a1 ?b))
    (at start (not (connected ?a2 ?b)))
  )
  :effect (and
    ...
    (at start (not (connected ?a1 ?b)))
    (at end (connected ?a2 ?b))
  )
)
```

5.2.3. Turns

A train driver must sit at the front of the train when moving. This means that if the train has to move in the opposite direction, the driver must stop the train, walk to the other side, and transfer control there. This is also called performing a saw-movement by van den Broek [55], and van den Broek et al. mention in [54] that this can take anywhere from 3 to 5 minutes, depending on the train type.

Kamenga et al. [2], who call it turnarounds, mention that they can be included in the paths taken but do not explicitly state what the effect is on the path duration. The additional durations of turns are acknowledged by Cuilenborg [14] but are ignored in their approach to solving.

In our domain, we keep track of the direction of a train with two literals, `direction_aside` and `direction_bside`, which are set to true if a train driver is sitting in the A-side cockpit or the B-side cockpit, respectively. A `turn` action can only be performed if a train is `operated` by a `driver` object. This action only changes the direction literal of the train, but it does make the `driver` and `train` objects unavailable for other actions during the turn.

The inclusion of this action prevents us from underestimating the durations of movements by always ensuring that the train's direction is set before it moves in that direction. It is worth mentioning that the same effect can be achieved by having a driver exit a train and enter from the other side. However, this takes longer than performing a turn, which justifies the inclusion of specific turn actions because it allows us to make more accurate predictions for the total duration of a plan.

5.2.4. Coupling and Uncoupling

The act of coupling combines two trains to form a single longer train, whereas uncoupling splits a train into two smaller ones. These two actions are the only ones that allow a planner to solve the **matching** sub-problem. Many approaches, such as those by Freling et al. [19], solve it as a separate problem first or combine it with the **parking** sub-problem, as done by Kroon et al. [31]. Kamenga et al. [2] consider all possible configurations of train units and only decide how to solve the **matching** sub-problem after solving the others first. In these approaches, they first decide which train units will be connected at the time of departure and later determine when and where they will be combined. The most common strategy is to perform all necessary uncoupling actions after the trains arrive and conduct all the coupling actions right before departure. Although often preferred, this strategy is not necessary. Train units can be coupled or uncoupled on many tracks as long as a driver is present to do so.

In our domain, we place slightly different restrictions on coupling and uncoupling actions. It is only possible to perform these actions while both trains are located next to one another on the same parking track and a driver is present. This means that it is allowed to couple or uncouple trains at any point during shunting, and any strategy regarding matching is the decision of the planner.

To verify in an action's predicates that both trains and the driver are located on the same track, we use the literals `train_at` and `driver_at`. We recall that we keep track of every train's exact location using `aside_distance` and `train_length`. These will be used to ensure that there is no other train between the two trains that are about to be coupled. This verification is not necessary for uncoupling, as a train, by definition, cannot contain another train.

When two trains combine into one, we could theoretically keep track of them and ensure that they always move together and remain next to each other. This is extremely complex to achieve in PDDL. Fortunately, we can take a slightly more elegant approach by introducing the concept of a `train` object being `active` or not.

In reality, when two trains combine, one could view it as the original two trains 'disappearing' and being replaced by a new train that just 'appears'. In our model, we view it as one of the two trains 'transforming' into the new train, whereas the other train 'disappears'.

When two `train` objects combine, we increase the `train_length` fluent so that it is the sum of both. The fluents used to keep track of the location of the train are updated to match the size of the newly combined train. The other `train` object 'disappears' by setting the literal `active` for it to false. Any action on a train can only be performed on one for which the `active` literal is true. This means that the inactive train does not participate in the state and essentially has disappeared.

When performing an `uncouple` action on a `train` object, it needs a second, inactive train as a parameter. In the effects of the action, the original, longer `train` object is shortened, and the remaining length is properly distributed to the other `train` object, which is activated. Now actions can be performed on the newly appeared train.

One last thing to keep in mind is that not every type of train can be combined with any other. In order

to verify this, we introduce the fluent `train_type_x`, where x is the name assigned to the type, to ensure that only the correct combinations are made or the correct splits occur. Although PDDL objects are typed, it is not possible to change them when solving, and thus this less elegant method using literals is required. This has a negative impact on the model's readability because we either have to write a specific action for every possible coupling or write one coupling action with many disjunctive preconditions and conditional effects. In our models, we choose to write out multiple actions, as we find this the most readable.

With these rules, we can correctly model (un)coupling with relatively few additional objects and actions. However, due to the lack of dynamic typing and the impossibility of dynamically creating objects, PDDL is not as suited for modelling the matching sub-problem. A snippet of the code is shown in Listing 5.4.

Listing 5.4: Updating literals to ensure correct coupling by PDDL planner.

```
(:durative-action couple_1_1_to_2
    :parameters (
        ?t1 - train
        ?t2 - train
        ?r - track
    )
    ...
    :condition (and
        ...
        (over all (train_at ?t1 ?r))
        (over all (train_at ?t2 ?r))
        (at start (= (+ (aside_distance ?t1)(train_length ?t1)) (aside_distance ?t2)))
        (over all (parking_allowed ?r))
        (over all (operated ?t1))
        (at start (train_type_1 ?t1))
        (at start (train_type_1 ?t2))
    )
    :effect (and
        ...
        (at start (not (active ?t2)))
        (at start (not (train_type_1 ?t1)))
        (at end (increase (train_length ?t1) (train_length ?t2)))
        (at end (train_type_2 ?t1))
    )
)
)
```

5.2.5. Marking Sub-Problems Solved

To ensure that the states contain sufficient information for determining the solvability of all sub-problems, it is essential to adopt a systematic approach. Early methodologies, solve multiple sub-problems sequentially. A more recent approach, considered the state of the art was proposed by van den Broek et al. [54] which integrates all sub-problems.

If our detailed modeled model can provide the same integrated solutions we must carefully examine how we recognise when sub-problems have been solved. The search algorithm explores various train routes across the shunting yard, identifying configurations in which all train units specified in the servicing schedule are serviced, and the appropriately configured trains secure valid parking spots. This subsection details how the PDDL model identifies the resolution of all sub-problems.

We commence by examining the **servicing** sub-problem. To ascertain whether a train has been serviced, we introduce the `serviced` literal. This literal is assigned a true value for a train if a corresponding `service` action has been executed with that train as a parameter. Such actions may only be conducted on designated servicing tracks, identifiable by the model through the `service_allowed` literal being set to `true`. For any train not requiring servicing during shunting, the `serviced` literal is set to `true` in the initial state. The **servicing** sub-problem is considered resolved when the `serviced` literal holds true for all `train` objects that are `active`.

In parallel with the servicing sub-problem, we introduce a `parked` literal for each train. The **parking** sub-problem is regarded as resolved when this literal is `true` for every active train. However, the mechanism

for establishing this literal differs significantly from the `serviced` literal.
In the domain, we incorporate `park` actions to facilitate the achievement of a valid parking configuration. A valid parking configuration is defined as one that allows for the smoothest possible train departures. Specifically, this means that each train can traverse the shortest path to the yard exit without being impeded by any train scheduled to depart later. We can confirm that train B does not obstruct train A if either it is on the same track but located further away from the entry track, or it occupies a different track with a greater `entry_distance` value. The entry distance is defined as the number of tracks that a train must move across to reach the entry track. If each train is unobstructed by any train scheduled for subsequent departure, this confirms that the departure order can be executed as smoothly as feasible.

It is important to highlight that, as specified in the problem definition, the departure schedule does not detail specific train units; rather, it specifies train types along with their respective departure times. In our models, we employ literals to classify trains, necessitating that we enumerate the parking actions for each type.

In the `precondition`s for all `park` actions, we implement a validation to ensure that the `serviced` literal is true for both trains, and that they possess the correct type. The goal state specifies that all active trains must be parked, which can only occur if the correct matching has been established. The actions selected by the planner are restricted to valid paths that do not result in crossings. When the `parked` literal is set to true for all active trains, this indicates that both the **matching** and **routing** sub-problems have been resolved.

Through the definitions of the `service` and `park` actions, we assign values to the `serviced` and `parked` literals for active trains. Once these values are established, we explicitly denote that the **servicing** and **parking** sub-problems are resolved. These literals can only be established once every train is correctly matched and has traversed a valid path, thereby confirming that their establishment also implies the resolution of the **routing** and **matching** sub-problems. A code snippet summarising the assignment of these literals is presented in Listing 5.5.

5.3. Simplified variations

In the previous section we have presented how we can describe TUSS in PDDL such that the resulting plan provides a solution to all sub-problems and guarantees feasibility if we ignore rules regarding the crew's working hours. A potentially negative consequence of this detailed description is that plan length, state size or the large number of available actions may lead to a bad performance by planners. In this section we explore how we can simplify the detailed domain such that we decrease the size of each state and limit the decisions a planner has to make. We do this hoping that these changes lead to better performance compared to the detailed domain.
We attempt to achieve these simplifications by formulating certain TUSS aspects differently or ignore them altogether. We realize that the obtained plans obtained do not provide the same guarantee of feasibility when compared to the detailed plans and thus need to be modified to improve them.
Below we describe several simplifications we consider. We will create domains containing different combinations of the proposed simplifications as to allow for comprehensive analysis of these simplifications.

5.3.1. Ignoring concurrency

Previously we have discussed how with the inclusion of $:durative\text{-}actions$ the planner is aware of what actions can be execute concurrently and provide a fully-ordered solution to the TUSS sub-problems. However, it might be worthwhile to consider creating models without this requirement as this opens up the possibility of using other planning algorithms as not all planners support the $:durative\text{-}actions$ requirement. The drawback of this is that the plans cannot provide fully ordered solutions for problem instances with multiple trains. The partially ordered plans thus require additional modification before they can be executed in a real shunting yard. This strategy where obtaining a PDDL plan is an intermediate step is seen in other research as well. One example of this is the paper by Estivill-Castro et al [17], where classical domains are used among other steps to solve a path-finding problem for a dynamic environment.

Planners will try to minimize the total duration of plans for temporal problem instances. Non-temporal

Listing 5.5: Where the parked and serviced literals are set.

```
(:durative-action service
    :parameters (
        ?t - train
        ?r - track
    )
    ...
    :condition (and
        ...
        (over all (train_at ?t ?r))
        (over all (service_allowed ?r))
    )
    :effect (and
        ...
        (at end (serviced ?t))
    )
)
(:durative-action park_different_tracks_0
    :parameters (
        ?t1 - train
        ?t2 - train
        ?r1 - track
        ?r2 - track
    )
    :duration (= ?duration 0.1)
    :condition (and
        (over all (train_type_2 ?t1))
        (over all (train_type_1 ?t2))
        (over all (train_at ?t1 ?r1))
        (over all (parking_allowed ?r1))
        (over all (parking_allowed ?r2))
        (over all (serviced ?t1))
        (over all (serviced ?t2))
        (at start (<= (entry_distance ?r1) (entry_distance ?r2)))
    )
    :effect (and
        (at end (parked ?t1))
        (at end (parked ?t2))
    )
)

(:durative-action park_different_tracks_1
    :parameters (
    ...
    )
    :duration (= ?duration 0.1)
    :condition (and
        (over all (train_type_1 ?t1))
        (over all (train_type_1 ?t2))
        (over all (parked ?t1)
        ...
    )
    :effect (and
        (at end (parked ?t2))
    )
)
```

domains, those without the :*durative-actions* requirement, can be instructed to either minimize the number of actions in the resulting plan or optimize the value of a metric. For our TUSS instances we believe the minimizing a cost metric will result in better plans because not all actions have the same impact on the total duration of the shunting plan. Entering and exiting a train takes significantly longer than moving a train across a track. Moreover, for trains that have to be serviced it does not make sense to punish the inclusion of `service` actions in the plan.

When optimizing for a metric term we need to include the :*action-costs* requirement in the domain and define a :*metric* term in the problem instance. With the :*metric* term it is possible to specify a

fluent to be either minimized or maximized. For TUSS domains it makes the most sense to $minimize$ a single numeric fluent called `total-cost`. The value of this cost is increased in the effects of actions, by how much depends on the impact an action has on the total duration. Actions that must be performed to reach a goal state will not increase this metric term.

5.3.2. Inexact train locations

We have described how we keep track of every train's exact location for the detailed model. Other TUSS solving methods, such as the current state of the art algorithm by vd Broek et al [54], take a different approach. An alternative would be to only keep track of a train's relative position on a track rather than precise coordinates. With this approach it is still possible to ensure that track capacity is never exceeded. The drawback is that there is no guarantee that the resulting plan is complete as it might be possible that additional `move` actions need to be inserted to ensure that no crossings occur.

To keep track of a train's relative location we introduce a numeric fluent called `order` which takes one `train` object as parameter. The smallest value possible is `1`, which is assigned to the train closest to the A-side end of the track. The largest value possible is the amount of trains that are on the track, which is stored under `num_trains`. The train for which its `order` equals the `num_trains` for the track its on is the track closest to the B-side end of the track.
To ensure that track capacity is never exceeded the `track_length` fluent must be dynamic, decreasing anytime a train enters the track and increasing anytime a train leaves it. For domains with inexact train locations the `track_length` fluent represents the available space on a track rather than its dimensions.

5.3.3. Ignoring turns

We have already expanded on the impact of turning a train has on plan duration and what the benefit is of including `turn` actions in the domain. It has been seen in other research, such as Cuilenborg [14], that no distinction is made between the impact of turning a train and the impact of moving a train. In our domains we take it a step further where we do not require that a train be reversed before moving in a different direction than previously.

This can be achieved simply by not including any `turn` actions or any $preconditions$ or $effects$ that contain any literal describing the direction of a train. Models containing this simplification contain less literals in each state, have on average less available actions in every state and will produce shorter plans. This means that less decisions have to be made by the planner to compute a plan for these domains.

5.3.4. Excluding switch actions

It is often seen in TUSS literature that movements are planned such that all tracks and switches between a starting point and destination are ”reserved” to prohibit use of them by other trains during the movement. This is how movements are planned by operators at NS and is modeled by vd Broek et al [54] as well as Kamenga et al [2]. This, combined with the assumption that the time it takes to move a switch is negligible, does not require them to model the action of moving a switch and allows the model to include multiple connections on each side of a track.
In our domains movements are included in a plan from one track to one other rather than long paths, which necessitates the inclusion of dynamic track connections to guarantee that a resulting plan does not cause crossings. We consider a simplification where track connections are static while realizing that this does not provide the same guarantee as other research regarding the impossibility of crossings.

This simplification is achieved by excluding the `potential_connection` literal from the domain and for every pair that this literal was true in the initial state we assign this truth value to the `connected` literal. Furthermore all `switch` actions are removed as well as any so-called phantom tracks included in the shunting yard description in the problem file. The result of these modifications to the domain is a reduction in state size and amount of decisions to be made by a planner.

5.3.5. Excluding drivers

Many other approaches do not consider the actions a driver takes as separate from a train movement, and they are often planned after a shunting plan has been created.

The exclusion of drivers might negatively impact the performance because the planner does not realize the true impact of specific action sequences. Consecutive movements of the same train take less time than sequences where the moving of trains is alternated. However, domains that do not include driver objects contain less action definition because actions such as a driver entering or exiting or turning a train do not need to be included. This should significantly decrease the decisions a planner needs to make to find a feasible plan.

Another benefit of including driver objects is that these infuse some resource management by the planner. Driver objects are a limited resource in the model and are required to be associated with a train before the train can move. The actions and literals related to driver objects allow for the correct modeling of which actions can be done concurrently and which cannot. For example, if there is only one driver object in a problem instance this means that at most one train movement can be performed at a time. To ensure this in a domain without driver objects we must introduce two fluents, which we call `max_concurrent_movements` and `concurrent_movements`. Before a `move` action is performed the predicates verify that `concurrent_movements` are less than `max_concurrent_movements` and only during the movement the `concurrent_movements` fluent is incremented by 1.

5.3.6. Excluding walking actions

Although walking durations are acknowledged as significant by vd Broek et al [54], crew planning is often treated as a separate problem. We also consider domains in which the walking times of drivers are ignored.

To achieve this we remove the `walk` action and the `walking_duration` fluent. This will reduce state size and hopefully make it easier for planners to decide on how to include movement actions to reach a goal state.

5.3.7. Combining actions

Some actions included in a shunting plan should only be followed by a specific other action. One example is that a driver should only enter a train if he intends to start moving it. If the driver were to exit a train right after entering we are in the same state as before and have wasted two expansions to get to that realization.

In PDDL however, it is not possible to make mandatory chains of actions. One way this can be modeled is to combine them into one action. This ensures that mandatory actions happen after one another but does mean that there are more grounded actions after parsing which complicates the search for the planner. This may cause complications during search which might have a negative impact on performance. However, if planners do not correctly recognize that some action combinations do not make sense a lot of time could be wasted exploring symmetrical states.

5.3.8. Duplicate sub-goals

As described before, the `parked` can only be set by actions which include the `serviced` literal in the preconditions. Thus in the goal state we do not need to mention the `serviced` explicitly, but it may be worth to investigate whether this has a positive impact on planner performance

5.3.9. Strict parking preference

In order to be able to describe when a train is located in a valid parking spot we introduced the `entry_distance` fluent. With this fluent we describe how many other tracks there are between a track and the exit of the shunting yard. When a shunting yard contains many tracks that have the same value for `entry_distance` situations may arise where they are interchangeable when deciding on a parking configuration. A simple example is shown in figure 5.4 which describes a problem instance in which train 1 needs to depart the shunting yard before train 2. Both configurations are equally valid seeing

as the `entry_distance` of both tracks is the same.

In vd Broek et al [54] it is mentioned that human shunting planners prefer fixed patterns in their shunting plans. In an attempt to introduce this in our PDDL we consider variations where parking preferences are more strict. This is achieved by making sure that every numeric value for `entry_distance` is unique. When a set of tracks have the same value for `entry_distance` each of them is increased slightly with the difference being different every time. This has as a result that although when starting from any of the tracks a train needs to cross the same amount of other tracks to reach the shunting yard exit, when deciding on when to park there is a clear order of which track is considered closest and which track is considered farthest. This means that the illustration in figure 5.4 only contains one valid parking configuration. One of the tracks will have a smaller `entry_distance` than the other which means that train 1 must be parked on this track specifically. This modification does not change the state size or the amount of actions included in plans. Further analysis is needed to predict how this impacts planner performances.

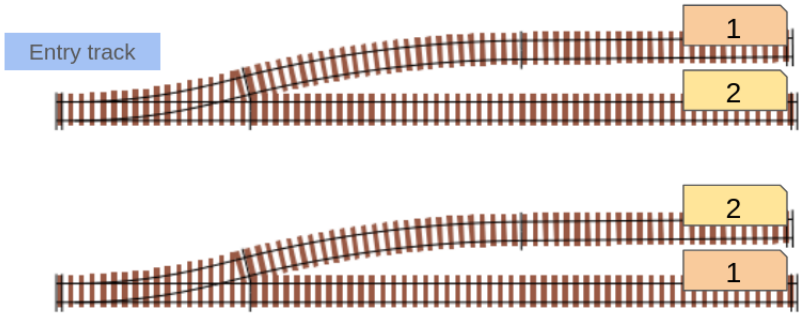

Figure 5.4: Two different parking configurations in which train 1 has to depart the yard before train 2 does. Without strict parking preferences both configurations are valid whereas with strict parking preferences only one of them would be.

5.4. Post-processing

Plans obtained from some of the simplified PDDL models lack certain details. In addition to a partial order solution for the parking, routing, and service sub-problems, they may omit several aspects, such as actions performed by drivers or actions describing the turning of a train.
Given the movement and service actions for each train and the partial order in which they should occur, a Constraint Programming (CP) model can be formulated in MiniZinc [43] to find the optimal way to schedule the missing activities. The resulting plans will contain `movement`, `service`, `enter`, `exit`, `turn`, and `walk` actions.
Our post-processing routing can be performed on solutions obtained for any of our PDDL models. In this section, we first explain the information extracted from the PDDL plans and how it is used to formulate the input for the CP model. Next, we describe the contents of the CP model.

5.4.1. Parsing PDDL plans

The first step of our post-processing routine is to extract the information from the PDDL plans necessary for the CP model.

Extracting relevant actions

The first part of our post-processing routine involves a script that extracts relevant information from any plan found using our PDDL models. While not all models contain the same types of actions, all models include movement and service actions. A script is used to extract these from any plan, regardless of the amount of detail it contains.

Ordering actions by train and track

The simplified plan is treated as a partial-order solution. The script recognises which actions must be performed before another by applying the following rules:

1. For each train, the order of actions performed by it must remain the same.
2. For each track, the order of actions that have the track in their parameters must remain the same.

The script extracts pairs of actions for which one must finish before the other can start according to these rules and saves the pairs in a list of tuples.

Recognising turns

The next step is to recognise which actions require a `turn` action between them. This is easily identified by examining the movements per train, and anytime two subsequent movements are in different directions, these actions are saved in a list of tuples.

Recognising switches

The penultimate step is to recognise which pairs of actions require a switch action to occur between them. The only rail infrastructure included in the PDDL models consists of tracks, and our script models connections between tracks dynamically, much like this is described in sub-section 5.3.4. The important rule to follow is that every track can be connected to at most one other track on both sides.
Before iterating through pairs of actions, the script assumes no connections are present between tracks. During the iteration, when a movement is encountered, the script checks whether a connection already exists for the sides of the tracks between which a train is supposed to move. If not, the connection is saved. If it exists and does not allow the second movement in the pair to occur, this pair of actions requires a switch action in between them. The pair is saved in a list of tuples.

Building walking distance matrix

The final step in parsing the PDDL plans is to organise the walking times between actions. Obviously, walking can only occur from one location to another, not between actions. Therefore, we examine the actions and formulate constraints that after one action has finished, there must be at least enough time for a driver to walk from that action's destination to the starting point of the next action.
The procedure to achieve this is straightforward: we first initialise a square matrix with the lengths of both dimensions being the number of movements. We label the actions $1..N$ and fill in the matrix at index (i, j) with the walking time from the destination of action i to the starting point of action j.

5.4.2. Constraint Programming Model

The CP model is created in MiniZinc. The model consists of a .mzn file in which all the constraints are described, as well as the variables, constants, and their bounds. After parsing the PDDL plans, a .dzn file is generated in which all the lists of tuples and bounds are formulated as constants. Together with the .mzn file, this can be executed to commence the optimisation of the parsed PDDL plan into a nearly complete TUSS plan.
In this subsection, we describe the variables and objective of the CP model, as well as the constraints.

Variables

Our CP model is an interpretation of a standard resource-constrained scheduling problem. In such a problem, there is a list of tasks that need to be performed as efficiently as possible. These are described by two arrays: one constant array containing the durations and one variable array containing the start times. The objective is to minimise the time at which the latest task finishes.
Similarly, in our CP model, we have a list of actions for which we have durations, and we need to find the optimal starting times. These arrays are called D and S, respectively.
However, in our CP model, we have one additional variable array in which we store the driver assignments. One of the constants in the model is the number of drivers, which is the limited resource in our optimisation problem. Every driver is labelled from $1..NUM\_DRIVERS$, and the CP solver determines the optimal driver assignment such that the total duration is the smallest possible. If the variable array containing driver assignments, or DA, is assigned the value 3 at index 8, this means that the action labelled 8 will be performed by driver 3.
To formulate the objective, we need to include a variable integer called END.

Objective

The END variable represents the time at which the plan concludes. To ensure END takes this value, we introduce the following constraint:

$$\forall_i S[i] + D[i] \leq END \tag{5.1}$$

This means that the latest timestamp at which any action can end is END. The objective is then set to minimise this variable integer, which causes the CP solver to find the most time-efficient schedule. The objective is formulated as follows:

$$\text{minimise}(END); \tag{5.2}$$

Constraints

Next, we formulate constraints so that the solution provided by the CP solver reflects the reality of operations within a shunting yard. The basis of our constraints includes the following assumptions about shunting:

- It takes 2 minutes to climb inside and start up a train.
- It takes 2 minutes to shut down a train and exit it.
- It takes 4 minutes for a driver already inside the train to start moving in the opposite direction.
- It takes 1 minute to move across any track.
- It takes 1 minute to change an electrified switch.

We acknowledge that these assumptions may differ slightly from actual operations. However, the most significant contribution is the solving *method*. These assumptions can easily be updated in the model if it were ever to be used for realistic shunting plans.

Precedence constraints

Using the parsing method discussed in 5.4.1, we extracted the pairs of actions for which these constraints will hold. For each pair of actions (i, j), the following constraint is formulated:

$$S[i] + D[i] \leq S[j] \tag{5.3}$$

This constraint indicates that the earliest action j can start is after action i has been completed. The best-case scenario occurs when actions i and j describe movements in the same direction, performed by the same train and driven by the same driver. In that case, these actions must occur sequentially, but there does not need to be any delay. In other cases, where the actions do not have the same direction, for example, constraints are formulated such that the difference in end and start times must be greater.

Turn constraints

As mentioned in the previous sub-section, two movements of the same train cannot occur immediately after one another, even if they are driven by the same driver. The action pairs requiring turn constraints are extracted using the method described in 5.4.1. For each such pair (i, j), the following constraints are written:

$$S[i] + D[i] < S[j] - 3 \tag{5.4}$$

Here, it is stated that the start time of action j must be at least 4 minutes after action i finishes.

Switch constraints

The last of these similar constraints applies to the pairs that require a switch action in between, as extracted by the procedure in 5.4.1. For each pair (i, j), we write the following constraints:

$$S[i] + D[i] < S[j] \tag{5.5}$$

This indicates that the start time of action j must be at least 1 minute after action i completes, which is the time required to move an electrified switch.

Driver changing trains

If a shunting plan specifies that a driver needs to move one train first and then another, they must exit the train they are currently in, walk to the other train, and start it up. For each pair of actions (i, j) that involve different trains but the same driver, we write the following constraints:

$$(S[i] + D[i]) < (S[j] - 3 - \text{walk\_times}[i, j]) \vee (S[j] + D[j]) < (S[i] - 3 - \text{walk\_times}[j, i]) \tag{5.6}$$

This constraint ensures realism regardless of whether i or j is planned earlier than the other. In case i finishes before j: after movement i is completed, the driver requires 2 minutes to shut down $train_i$. Afterwards, they walk to $train_j$, which takes an additional 2 minutes to start that train up. These constraints describe the absolute minimum amount of time it takes a driver to change trains.

Train changing drivers

In the opposite case, where we have two actions i and j performed by two different drivers but on the same train, we formulate the constraint slightly differently:

$$(S[i] + D[i]) < (S[j] - 3) \vee (S[j] + D[j]) < (S[i] - 3) \tag{5.7}$$

In this instance, we only need to consider the start-up and shut-down time of the train, as this represents the best-case scenario for the fastest driver swap.

Redundant constraints

In the background, in Section 2.4, we mentioned that the inclusion of redundant constraints might improve performance. The redundant constraints included use MiniZinc's built-in `disjunctive` constraint, which specifies that two actions cannot overlap in time. Any action pair that has either a precedence, turn, or switch constraint is also assigned the redundant `disjunctive` constraint, which implies that these pairs cannot overlap in time.

Symmetry breaking constraints

There is a symmetry that we can eliminate regarding driver assignments. At the beginning of the shunting plan, none of the drivers are busy, and thus it does not matter which driver is assigned to the first task. We introduce a symmetry-breaking constraint that assigns driver 1 to action 1. As mentioned in Section 2.4, this induces the solver to search more efficiently and thus improves performance.

Choice of solver

Any general-purpose solver could be used to solve this described CP model. Any time we execute this model for this thesis, we utilise the Chuffed solver [12].

5.5. Conclusions

In this chapter we explored how a detailed model would model the necessary aspects of TUSS problem instances to be able to create feasible plans. Although this detailed model might not produce plans that are ready to use as is, it is possible to model problems with a similar level of detail as the current state of the art. Although it is possible to integrate all sub-problems in PDDL, its features are not perfectly suited to model the matching sub-problem.

Besides the description of a detailed model we offer numerous options to simplify this detailed model. In combination with our post-processing routine these models could produce plans with a high level of detail with potentially better performance.

The code for the post-processing routine is uploaded to Github [36].

6

Experimental analysis of planner and model performance

Now that we have explored various approaches to model and solve TUSS instances, we proceed to evaluate the performance of different planners and models through a series of experiments. These experiments are designed to address the following key research questions:

1. Which planner-model pair demonstrates the best overall performance?
2. Which simplifications, as discussed in Section 5.3, significantly impact performance?
3. Are there specific areas for improvement in the content or quality of the generated plans?

This chapter begins by detailing the experimental setup, followed by an in-depth analysis of the performance of various planner-model pairs. The chapter concludes with recommendations for the best-performing planner-model pair and suggestions for improving the analysed planning algorithms.

6.1. Experimental setup

To evaluate the performance of different planner-model pairs, we establish a well-defined experimental setup. This section outlines the models and planners that are compared. Moreover, we describe what problem instances are generated for each planner-model pair. Next we detail the steps performed to obtain the results and provide the metrics we use to evaluate them.

6.1.1. Models

Experiments are run for multiple models in which descriptions of various TUSS aspects are either detailed, as mentioned in section 5.1 or simplified, as mentioned in section 5.3. Because of the large amount of models evaluated we organise them in four categories to improve clarification. The categories are decided based on the two modeling decisions with the most significant impact on the way a problem instance is approached.
The most important the decision is whether to consider concurrency when planning, as described in sub-section 5.1.1, or ignore concurrency, as described in sub-section 5.3.1. With the inclusion of concurrency a temporal planner is required whereas a numeric planner suffices otherwise.
The second most significant decision is whether train locations are modeled exactly, as described in subsection 5.1.2, or relatively, as described in subsection 5.3.2. Exact train locations limit the number of available movements per state more than relative locations do.

The models are organized in four categories based on the outcome for the two aforementioned modeling decisions. How these categories are numbered is explained in table 6.1. Per category multiple variations are evaluated, a more detailed overview of all models used in the experiments is presented in the appendix in chapter A.1.

Table 6.1: Numbering of model categories based on the two most significant modeling decisions and how many variations were analysed in each category.

Category	Concurrency	Train locations	Total number of variations in this category
1	Included	Exact	19
2	Included	Relative	14
3	Ignored	Exact	18
4	Ignored	Relative	14

6.1.2. Planners

Over the years many planners have been developed to compete in International Planning Competitions (IPCs) [39]. Many of them however have not been maintained and as a result are no longer compatible with newer compilers and/or operating systems. In table 6.2 we provide an overview of the planners for which it was possible to solve our TUSS models and which model categories they are compatible with.

Table 6.2: Overview of planners and their properties. The column **Planner** lists the used planners: TFD is Temporal FastDownward [41], OPTIC is Optimizing Preferences and TIme-dependent Cost [5], ENHSP is Expressive Numeric Heuristic Search Planne [45], Metric-FF is Metric-FastForward [28], SP is the SymbolicPlanners.jl package [56] and NFD is Numeric FastDownward [3]. The column **Search Algorithm** lists the search techniques employed by each planner: A^* is Dijkstra's A-star search algorithm [24], WA^* is Weighted A-star, IDA^* is iterative deepening A-star [30], DKS_A^* is Dynamic K-Shortest Paths A-star [48] and EHC refers to Enforced Hill Climbing [27]. The column **Heuristic** specifies the heuristic functions used: h_{add} is the additive heuristic [8] and superscripts ^{ce} and ^{ir} refer to the context-enhanced [18] or interval-relaxation [46] modifications. h^{ff} is the Fast-Forward heuristic [28], h^{rpg} refers to the relaxed planning graph used in POPF [13], h^{mrp} is the multi-repetition plan heuristic [47]. The final column **Compatible Model Categories** refers to the categories of TUSS models these planners can handle.

Planner	Search Algorithm	Heuristic	Compatible Model Categories
TFD	A^*	h^{ce}_{add}	1, 2
OPTIC	WA^* + IDA^*	h^{rpg}	1
ENHSP	WA^*	h^{mrp}	3, 4
Metric-FF	EHC + WA^*	h^{ff}	3, 4
NFD	DKS_A^*	h^{ir}_{add}	3
SP	WA^*	h_{add}	3, 4

Both short descriptions of the algorithms used by the planners and a more comprehensive overview of the PDDL features supported by each planner can be found in the appendix in section A.2.

6.1.3. Problem instances

To evaluate their performance, each planner-model pair will attempt to solve multiple problem instances with varying degrees of complexity. The complexity is measured by the number of trains included in the problem. From chapter 4 we recall that the problem description should include a shunting yard layout as well a scenario consisting of a arrival, departure and service schedules. In this sub-section we describe these properties for the instances used in our experiments.

Shunting yard layout

For all problem instances we use a slightly modified version of the Kleine Binckhorst shunting yard. This layout was used by other authors such as van den Broek et al. [54] and Cuilenborg [14]. In our experiments this layout is modified by removing one of two entry tracks, removing an inspection track and changing an exterior cleaning track to a regular parking track. After these modifications we are left with a shunting yard containing one entry track, 10 parking tracks and 2 service tracks.

The original layout of the Kleine Binckhorts shunting yard is presented in figure 6.1 and the modified version used for our experiments is shown in figure 6.2.

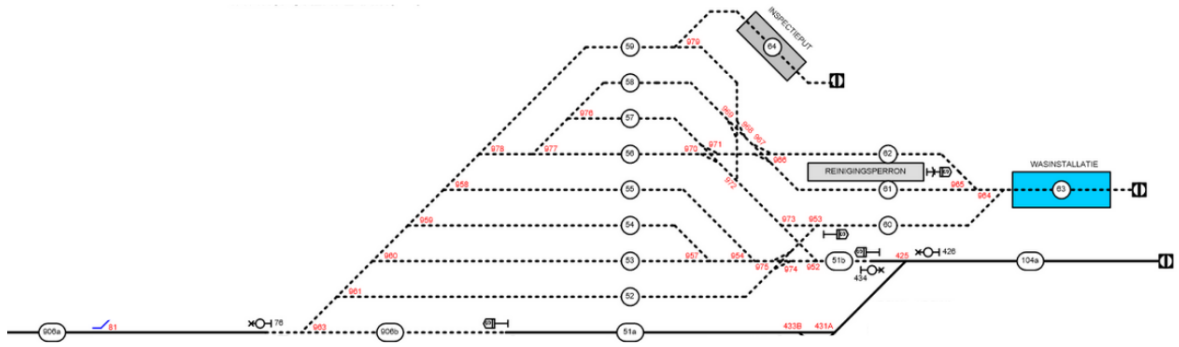

Figure 6.1: Layout of the Kleine Binckhorst, a real shunting yard operated by NS.

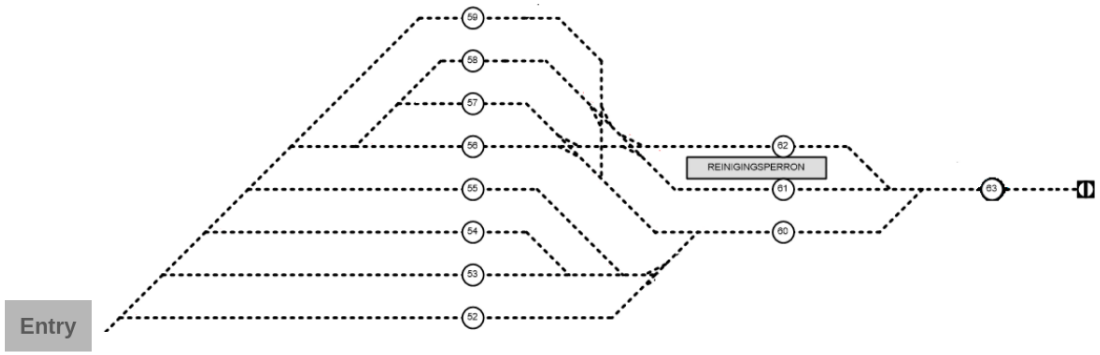

Figure 6.2: Shunting yard layout used in problem instances for experiments.

Scenario

For the scenario used we took inspiration from those used by NS to test their automated planning system. In our scenarios the entry schedule describes multiple trains of the same length arriving at the entry track simultaneously. All trains need to be serviced and must depart in the same order and composition they arrived in. This means that no coupling or uncoupling is to be done during shunting. A goal state is reached when all trains have been serviced and are in a valid parking configuration. A valid parking configuration is one where every train is able to travel across the shortest path to the exit without being blocked by any train that is scheduled to depart later.

Increasing complexity

The shunting yard layout and scenario is kept constant, thus the complexity depends on the amount of trains included in the problem. For every planner-model pair this number starts at 3 trains and increases up until the planner-model pair has failed 3 times. The shunting yard layout used in the problem instances fits a maximum of 32 trains so no planner can ever find a solution for instances containing more than 32 trains.

6.1.4. Method

Now that we have outlined the ingredients needed to perform an experiment we describe the process using the following steps:

1. Select planner, model and problem instance

2. Perform search after providing the planner with the model and problem instance
3. Obtain partial-order plan from planner output
4. Finalize solution with post-processing routine described in section 5.4

A time limit of 300 seconds is set for step 2 and if a plan could be obtained after step 3 another time limit of 300 seconds is set for the final step. If the planner could not produce a partial solution the experiment is considered to have failed.

The post-processing routine performed in steps 4 is described in more detail in section 5.4. These steps can be done on plans obtained from any of the TUSS models. The final solution obtained after post-processing always contains the same level of detail and allows for a fair comparison between different models containing different levels of detail.

6.1.5. Evaluating results

As mentioned before, the purpose of setting up these experiments is to be able to compare performance for different planner-model pairs, and in this section we define how we do so. When measuring the performance we look at the following properties in order of importance:

1. The complexity of problem instances for which a solution could be found
2. The makespans of obtained shunting plans

The most important measure of performance is how complex the instances are for which a solution can be found. The more trains included in the problem the more complex it is deemed. Being able to create shunting plans for more trains increases the capacity of a shunting yard which is of great interest for railway operators.
When comparing two planner-model pairs, we look at the most complex problem instance that could be solved. The pair that can solve the most complex problem is considered to have better performance.

If two planner-model pairs are able to solve similarly complex problem instances, we look at the quality of the plans they produced. In this paper, we compare plan quality quite superficially, looking only at which has the shortest makespan.
Human planners prefer robust plans to be generated. The makespan of a plan is considered a simple but commonly used measure for robustness by Boysen et al. [9] and van den Broek et al. [10]. In this paper, we naively only consider the makespan of a plan when measuring robustness. We consider one planner-model pair's performance better than another if it can consistently produce plans with shorter makespans for instances of the same complexity.

6.1.6. Hardware

All experiments were performed on an MSI Modern 15 B7M laptop with an AMD Ryzen 7730U processor and 16 GiB RAM. The operating system used was Ubuntu 24.04.1 LTS.

6.2. Comparing performance

In this section we present an analysis of the experimental results obtained with respect to performance. We compare different planners, as well as different model variations. First it is important to mention that the limiting factor for all experiments was the PDDL planner. The post-processing for all 410 experiments performed finished within 4 minutes.

Before we have identified the inclusion of concurrency and the modelling of train locations as the modelling decisions with the most significant impact. After these we consider the inclusion of driver objects to be the third most significant. The inclusion of driver objects necessitates actions to model them entering and exiting trains which significantly increases state size and te amount of actions necessary to complete a partial-order plan.
Next, the modelling of turning a train is considered the fourth most significant modelling decision. This greatly impacts how certain paths across the yard are evaluated.
In this section we first compare results per planner and based on the aforementioned modelling decisions. Lastly we provide a brief overview of the impact of the other simplifications that were deemed

less significant.

6.2.1. Comparing planners

Several planners were tested that take different approaches to producing solutions. All of them do some form of search guided by a heuristic evaluation. Below we compare which planner was best able to find feasible partial-order plans to the TUSS instances.

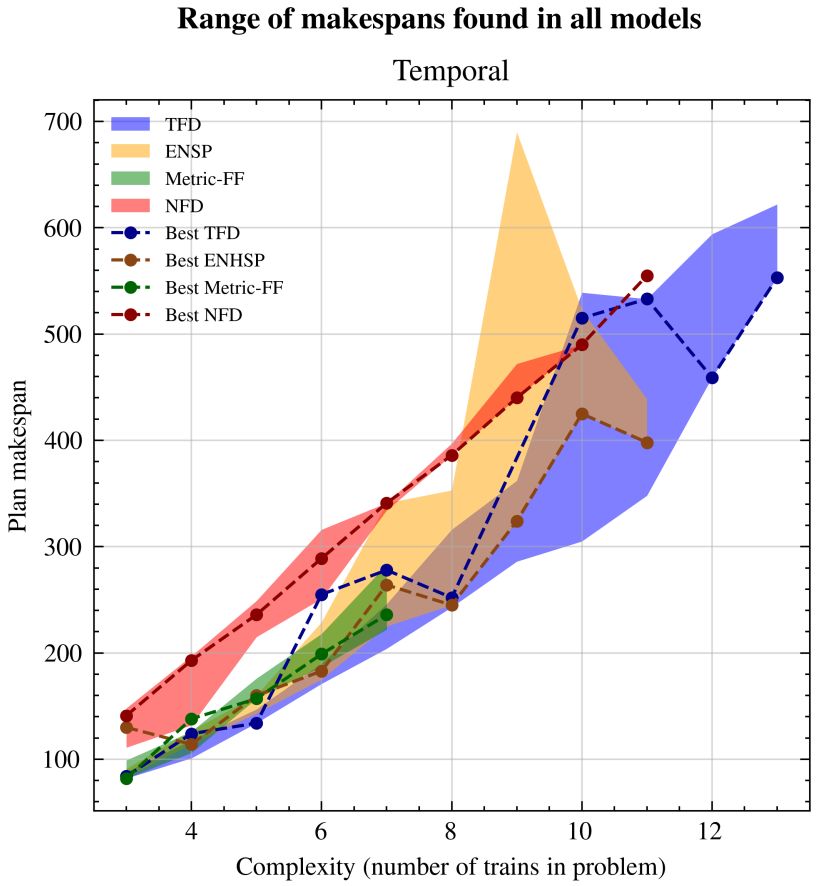

Figure 6.3: Comparison of performance between different planners. For each planner the area was filled between the best and fifth-best result found per level of complexity.

In figure 6.3 we plot the 5 shortest makespans per level of complexity for each planner. Two planners were omitted, OPTIC and SymbolicPlanners (SP), because they were never able to solve any problem instance with more than four trains. These planners are deemed unsuited for solving TUSS problem instances.

The best performing planner overall was Temporal FastDownward. It was able to solve the most complex instances and the shortest makespan for each level of complexity was always one found by this planner.
The best performing numeric planner was ENHSP. It was able to solve problem instances with up to 11 trains, and the models can produce plans with relatively short makespans compared to other partial-plans found. For instances with up to seven trains Metric-FF had a similar performance to ENSP, but interestingly Metric-FF was never able to solve instances that were more complex. Curiously that this was never because the planner timed out during search, search was never initialized for problem instances with more than seven trains.
After OPTIC and SP the worst performing planner was Numeric FastDownward (NFD). The best plan

found using this planner was always worse than at least five plans found by any other planner for the same problem instances if they contained eight trains or less. The difference in performance was smaller for more complex problem instances but the shortest makespan found by any model paired with NFD was still significantly longer than the shortest makespan found for any model paired with another planner.

Overall we can conclude that both TFD and ENHSP performed significantly better than the other planners, with TFD having the best performance.

The best planner-model pairs all include the TFD planner. There are two candidate models that can be considered to allow for best performance, all of which are presented in figure 6.4

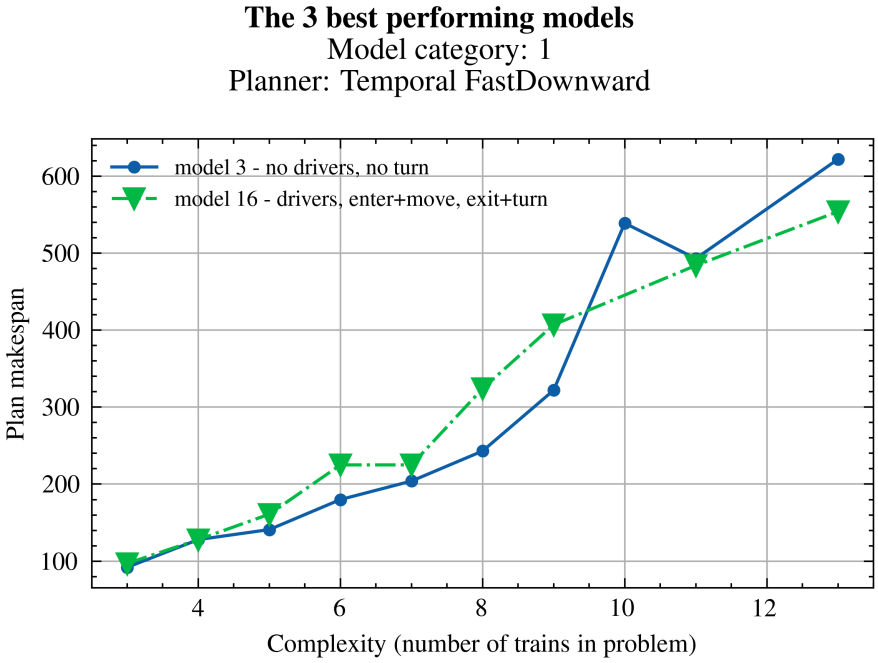

Figure 6.4: Top two candidates for best performing planner-model pair.

6.2.2. Temporal vs. Numeric

In section 5.1.1 we explain that in order to model concurrent actions we need temporal domains and planners, whereas otherwise numeric domains and planners suffice. All models in categories 1 and 2 are temporal, whereas all models in categories 3 and 4 are numeric.

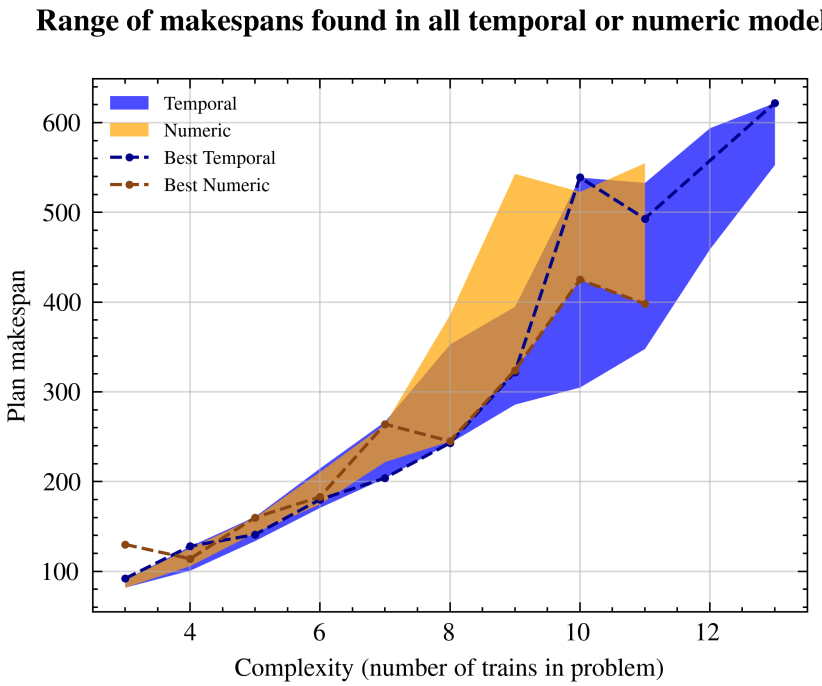

Figure 6.5: Comparison of performance between different temporal and numeric models. For each group the area was filled between the best and tenth-best result found per level of complexity.

In figure 6.5 we compare the ten best temporal and numeric results found per level of complexity. We can see that temporal planner-model pairs outperform numeric planner-model pairs because they were able to solve problem instances containing up to thirteen trains, whereas no numeric planner-model pair could solve any instance with more than eleven trains.

Even considering plan quality temporal planner-model pairs perform better. The best temporal pair per level of complexity always has a shorter makespan than the best numeric pair. The difference is more significant for more complex problem instances.

6.2.3. Exact vs relative train locations

Besides the modeling of concurrency, the formulations of train locations was considered the most significant modeling decision to be made. Modelling exact train locations can limit the number of available actions per state. We compare the impact of how train locations were modeled for the ten best planner-model pairs in either category in Figure 6.6 below.

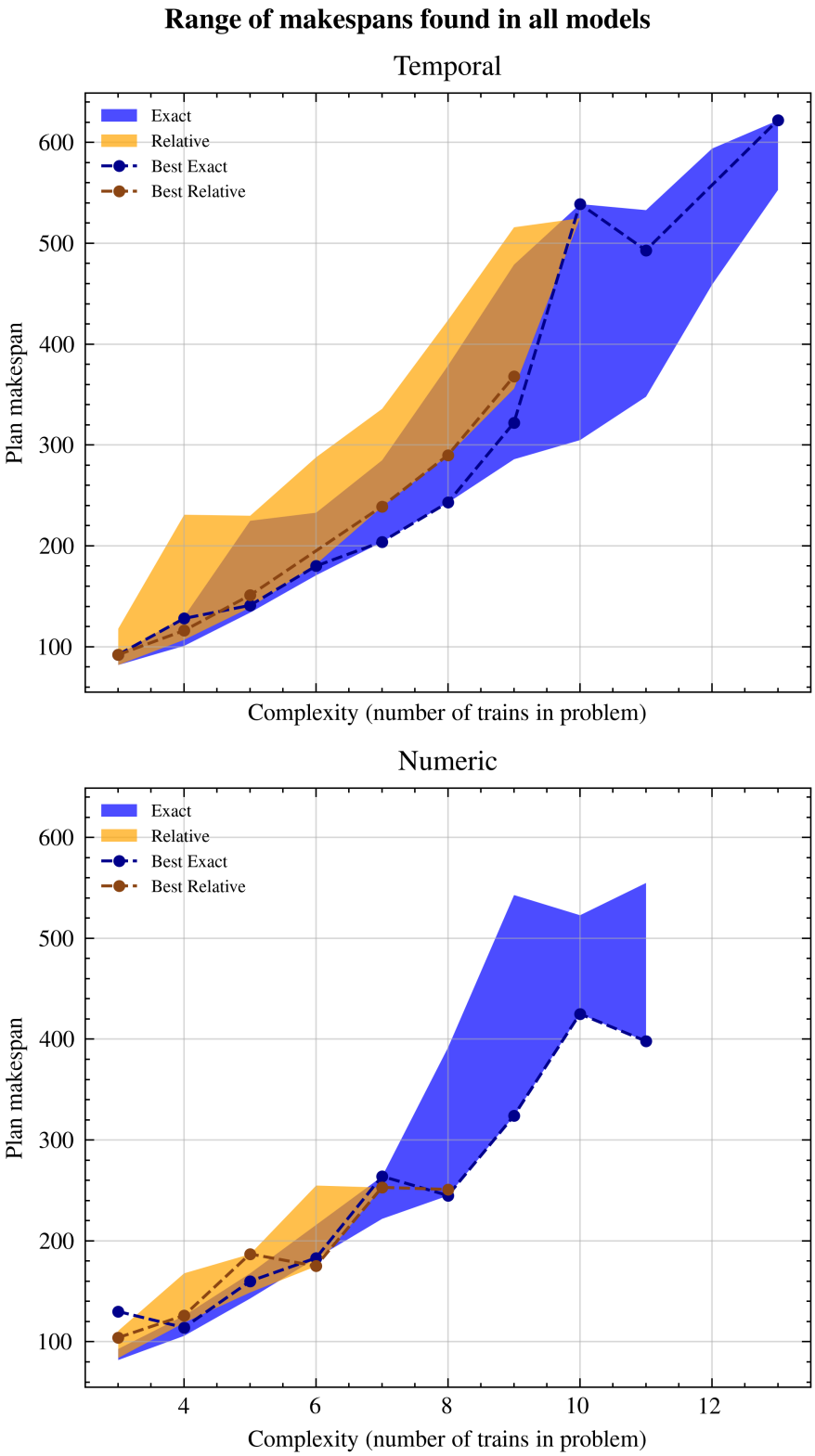

Figure 6.6: Comparison of performance between models where train locations were modelled exactly or relatively. For each group the area was filled between the best and fifth-best result found per level of complexity.

We can see that both temporal and numeric planner-model pairs perform better when train locations are modeled exactly. In both cases, the ten best planner-model pairs were able to produce plans for significantly more complex problem instances. For the temporal domains we can also say that the best

model per level of complexity with exact locations always outperformed the best model with relative locations. The same cannot be said for numeric domains.

In Section 5.3.2 the concern was mentioned that with exact train locations it occurred more often that potentially viable movements were not considered and that this could make it more difficult for planners to find feasible plans. This concern was proven invalid. On the contrary, modelling exact train locations led to the planners finding feasible plans more easily.

6.2.4. Including vs. excluding drivers

Similarly to how we compared models with or without turn actions we compare models including or excluding driver objects for the two best performing planners. Introducing driver objects in a PDDL model significantly increases the state size and number of actions included in a plan. Before we expressed concerns that this would negatively impact the performance.

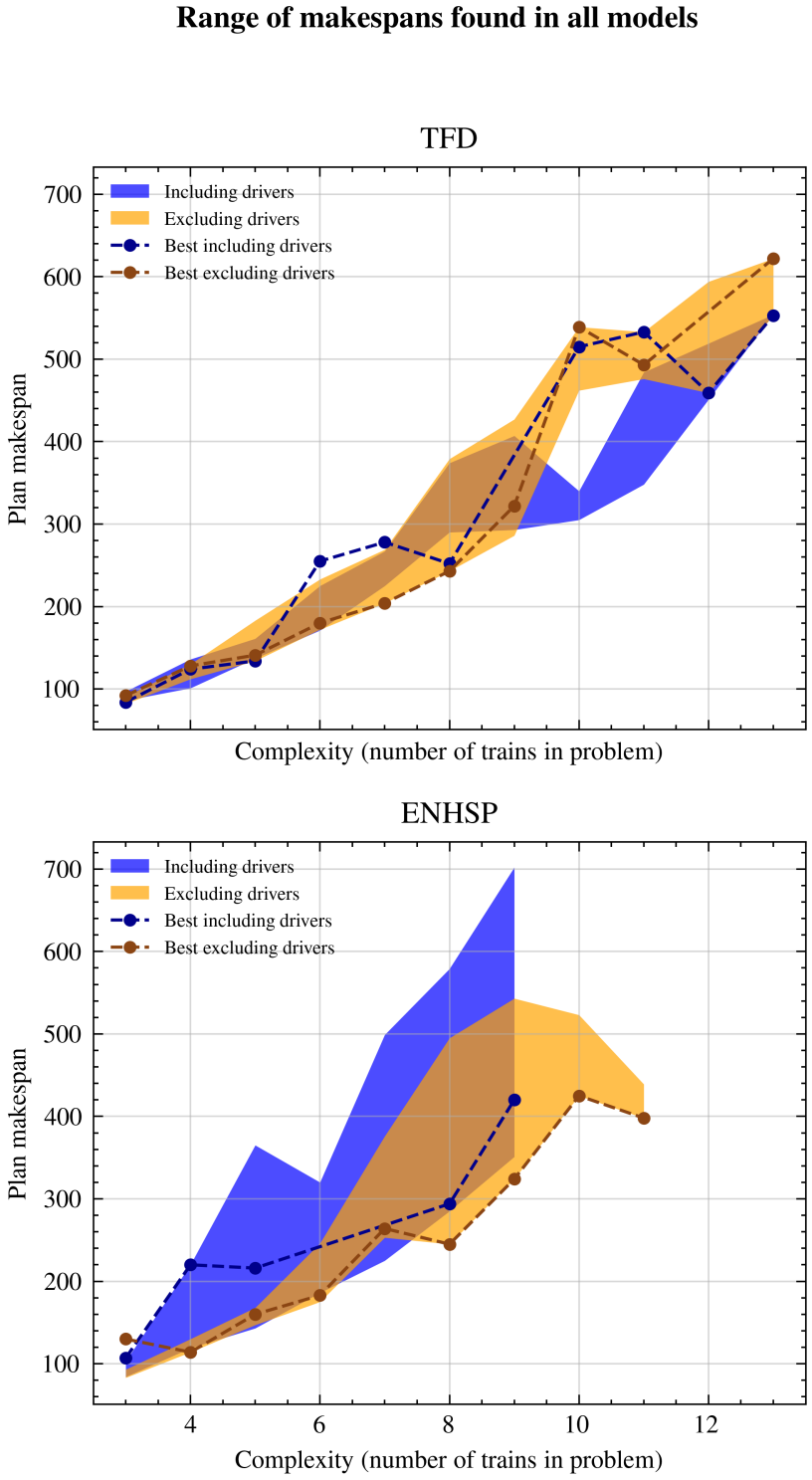

Figure 6.7: Comparison of performance between models including or excluding driver objects. For each group the area was filled between the best and fifth-best result found per level of complexity.

In Figure 6.7 we can see that these concerns were valid for the ENHSP planner. This planner is more likely to produce plans with shorter makespans and is able to handle instances with more trains if driver objects are not modelled.
However, for TFD the best models including drivers perform similarly to the best models without.

6.2.5. Including vs. ignoring turns

For each category several models were compared that included turn actions and more in which turn actions were ignored. In section 5.3.3 we expressed concern that without considering the additional time it takes to turn a train planners would end up producing plans with longer makespans. Below, in Figure6.8 we compare the impact of including turns in our PDDL models for the two best performing planners, TFD and ENHSP.

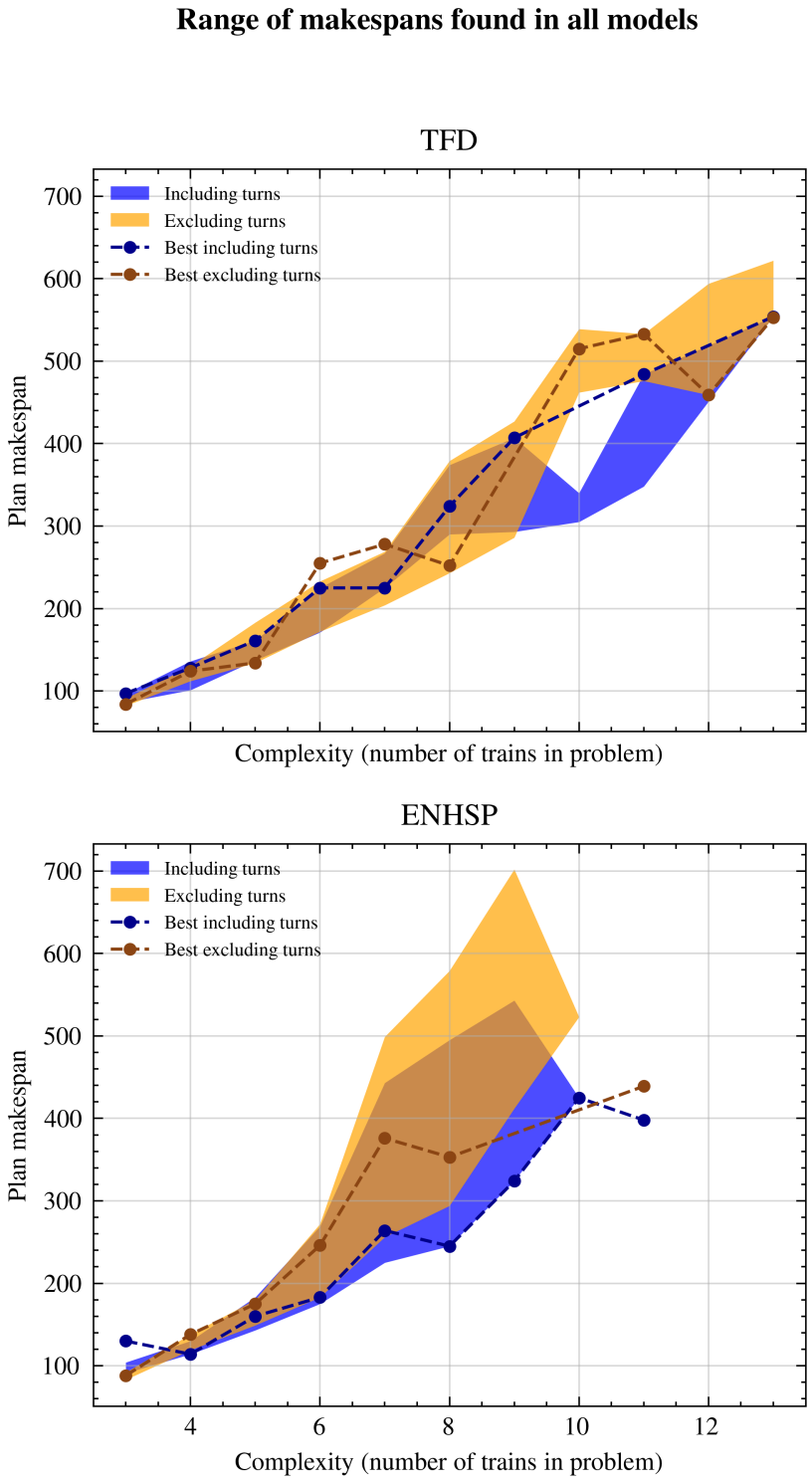

Figure 6.8: Comparison of performance between models including or excluding turn actions. For each group the area was filled between the best and fifth-best result found per level of complexity.

For both planners the inclusions of turns did not cause either of the planners to not be able to handle more complex models. For TFD the impact on the produced makespans was insignificant. The ENHSP planner produced was able to produce plans with the shortest makespans for models including turns. For ENHSP including turns does have a positive impact.

6.2.6. Other modelling decisions

The complete results of all the experiments performed are presented in the appendix in chapter A.3. The models used in experiments are uploaded to Github [37]. In section 5.3 several more simplifications are discussed and thus to keep this section organised we provide a compact overview of the overall impact of each simplification across all planners and model categories in Table 6.3

Table 6.3: A compact overview of the overall impact of several simplifications discussed in section 5.3.

Planner	Model Category	Excluding switch actions	Strict parking preferences	Excluding walk actions	Combining actions	Duplicate sub-goals
TFD	1	Positive	Negative	Positive	Positive	Neutral
	2	Positive	Neutral	Neutral	Neutral	Negative
ENHSP	3	Positive	Negative	Positive	Positive	Negative
	4	Positive	Negative	Positive	Neutral	Negative
Metric-FF	3	Positive	Negative	Neutral	Neutral	Neutral
NFD	3	Positive	Negative	Positive	Neutral	Neutral

The easiest simplifications to assess are the exclusion of switch actions and assigning strict parking preferences. Almost no planners were able to find any solutions for models including switch actions. Excluding switch actions greatly simplifies problems for all planners. The assignment of strict preferences for parking tracks can not be considered a simplification. This modification to models nearly always caused a worse performance compared to similar models without strict parking preferences.

In general the better performing planner-model pairs included models without walking actions. However, in for some planners this did not seem to have a significant impact. It can not be said that the makespans of plans obtained with models including walk actions were overall shorter. It does not seem that the inclusion of walk actions improves planner's understanding of the problem.
The combining of multiple actions is also hard to assess, as in some cases it improved performance and in some cases it worsened performance. There is not a combination of actions for which it can be said that it positively impacts performance in general.
Lastly, the inclusion of sub-goals at best did not have a significant impact on the performance of planner-model pairs. At best the performance was similar to models without.

6.3. Analysing PDDL plan contents

In the previous section we analysed the performance for the examined planner-model pairs based on problem comlexity and plan makespans. In this section we take a closer look at the contents of the plans produced by the PDDL planners and see if we can identify any areas where their understanding of the TUSS models seems to be lacking.
First, we examine if service actions are distributed optimally by the planners. Next, we examine whether the planning algorithms can accurately identify and eliminate symmetrical states from the search tree. We take a look at symmetries in state values and symmetries for which a more in-depth analysis of the TUSS problem instances is needed.
Next we propose a strategy that might improve performance, and analyse plan contents obtained in experiments to verify whether it is viable or not. Lastly we examine how search progresses in a model with strict parking preferences compared to a model without. The idea behind this simplification was to alleviate some decision-making from the planner and that parking tracks were assigned more consistently. With an example we try to understand better why this decision seemed to have such a negative impact.

6.3.1. Service track distribution

Service actions take significantly longer compared to other actions to be performed and therefore it is preferable to evenly distribute service actions across the available service tracks. While trains are being serviced either other un-serviced trains can be moved closer to the service tracks or serviced trains can be moved to their final parking destination. Below we present how evenly the service actions are distributed across both tracks.

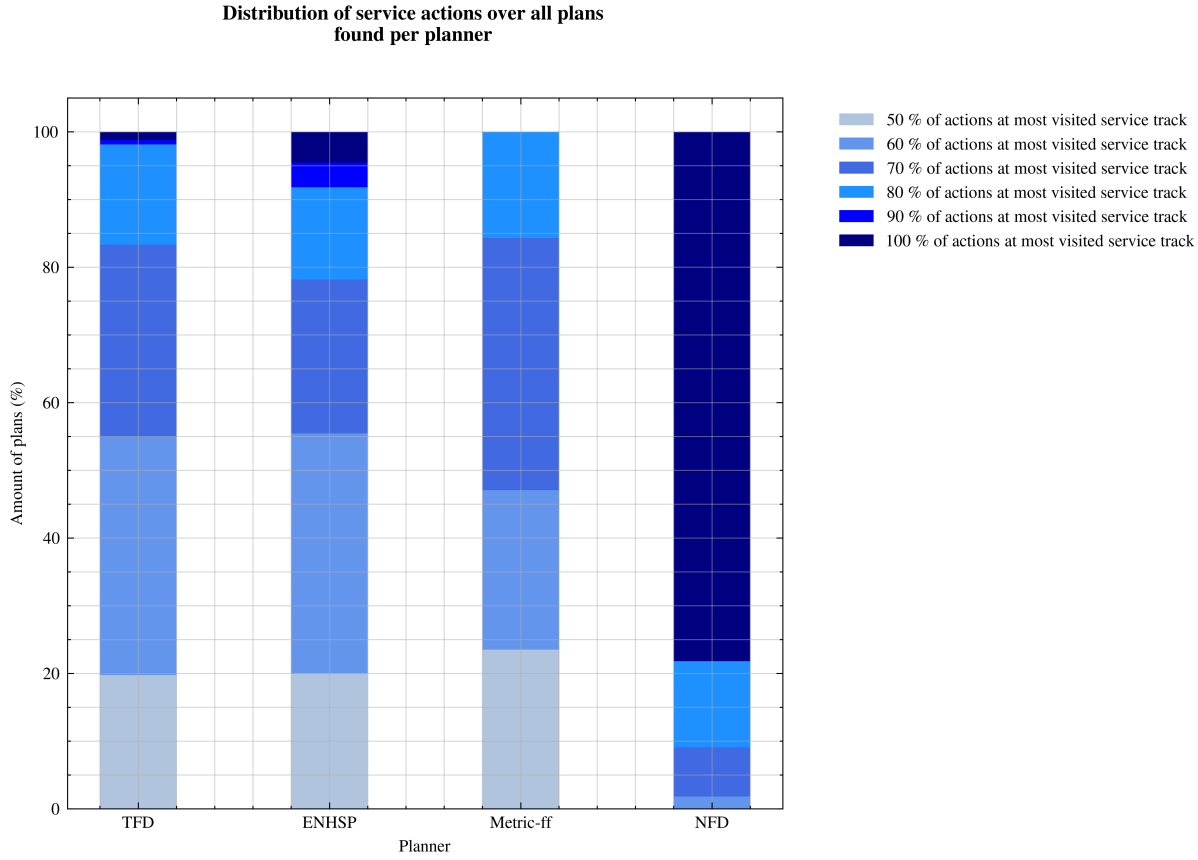

Figure 6.9: Analysis on how the locations of service tasks are distributed across the 2 available service tracks. The darker colours indicate that service tasks were very unevenly distributed, whereas light colours indicate even distribution. When looking at the height per colour we can see what percentage of the total amount of plans had which level of spread in their distributions.

In the numeric domains it is not indicated that servicing two trains simultaneously is preferred. Plans in which all trains are serviced on the same track have the same cost as those in which the service tracks are alternated. However, in figure 6.9 we can see that both ENHSP and Metric-FF distribute service actions as evenly as the only temporal planner used, TFD.
However, for nearly all plans obtained using NFD all service tracks are performed on one track. This explains the poor performance of this planner more clearly.
Important to note is that some models were created where an even distribution was forced for numeric planners, but for these models all numeric planners always were able to handle less trains compared to models without this specification.

6.3.2. Recognising state symmetries

First we discuss symmetries that are not specific to the examined TUSS models. In section 5.3.7 a concern is brought up that planners are unable to recognise when two states are symmetrical. Some models include an action together with its inverse. If the inverse of an action is performed after it we reach the same state. An efficient algorithm should only ever keep one node per state in its search tree. If it encounters a state it has already seen before, it should remove the node for which the cost

to reach it was largest. If both nodes can be reached with the same costs it should not matter which is kept in the search tree.

In this sub-section we examine the two best-performing planners, TFD and ENHSP and verify whether they recognise these symmetries. We examine two types of actions and their inverse, starting with $enter$ and $exit$ actions. When a train driver enters a train and exits it straight after we have reached a symmetrical state.
Since the code for the heuristics of ENHSP and TFD is compiled, it is hard to analyze whether this inefficiency occurs during search. Therefore we examined all solutions produced by these planners and counted when an $enter$ action was followed by an $exit$ actions for the same driver. The results of this analysis are presented in table 6.4.

Table 6.4: Amount of consecutive enter and exits pairs for models where this is possible.

Planner	Category	Model	enter followed by exit	total enters
TFD	1	9	4	460
		17	17	714
	2	8	0	187
		9	1	1001
		12	7	960
ENHSP	3	7	0	159
	4	9	0	30
		10	15	135

For both planners we could find instances where symmetrical states were reached and explored further. Anytime this happens for the ENHSP planner this leads to a plan with larger value for the cost. For TFD this is not necessarily the case, as the cost is measured by the makespan of a plan rather than how many actions are included. However, when examining plans obtained by this planner it was verified that in at least one case the execution of an $enter$ and $exit$ sequence lead to a plan with longer makespan.

Another such action pair that leads to a symmetrical state are the two different $turn$ actions. If a train turns to one side and then back to another nothing has changed in the state but the increase of the cost. In fact, $turn$ actions should only be followed by a movement, as exiting a train or servicing also does not make sense after a $turn$ action. In those cases the $turn$ action could be replaced by a gap in the plan and thus it is not preferred that planners explore states resulting from such a sequence. The results of how many times this occurs in plans for both planners can be seen in Table 6.5. It is quite surprising as it seems that for both planners it happens quite often that a $turn$ action could have been excluded from the plan. It should be mentioned that often the ENHSP planner does seem to rectify its mistakes and inconsequential turns are removed by the time it decides on a final plan. However, this finding this does indicate that time is spent exploring these symmetrical states, that should be spent on exploration of alternative action sequences.

Table 6.5: Amount of times a turn action was not followed by a move action for several models.

Planner	Category	Model	turn not followed by move	total enters
TFD	1	6	15	46
		7	30	75
ENHSP	3	4	81	143
	4	6	51	51

6.3.3. Recognising TUSS-specific symmetries

Besides symmetries that can be recognised from action definitions only, there is a deeper symmetry that follows from the shunting yard layout. In this section we examine the parallel paths that trains can take in the shunting yard layout and how the different paths could be treated more efficiently.

In our experiments we use a modified version the Kleine Binckhorst shunting yard in all problem files. The trains described all contain one unit that is 80 metres long. The layout described in all PDDL problem files can be seen in figure 6.10. In this figure we also note for each track how many trains they can fit at most.

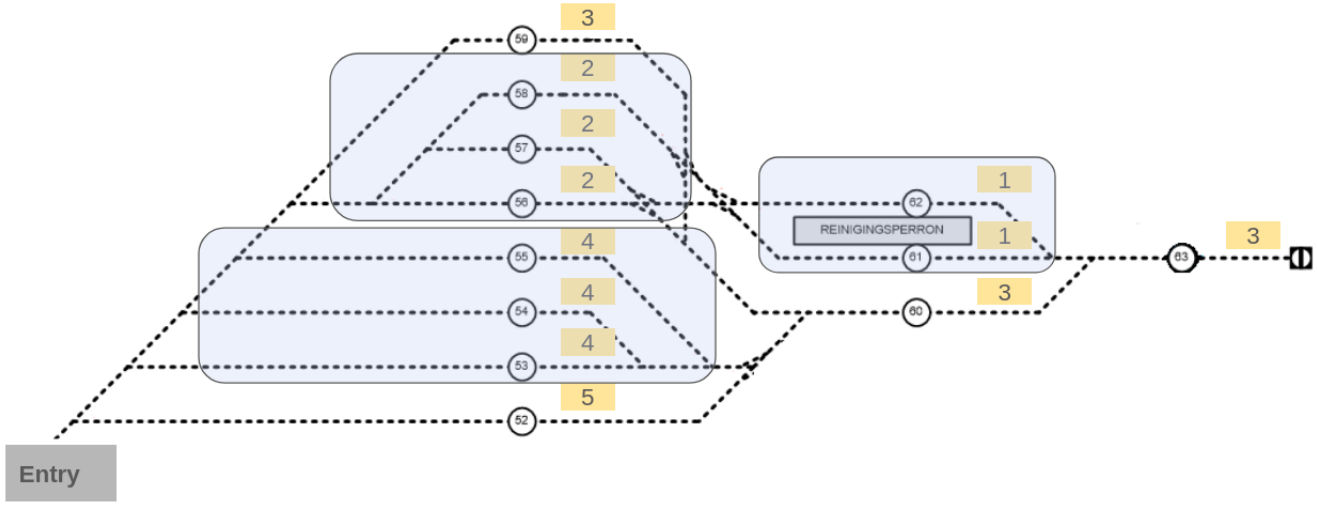

Figure 6.10: Shunting yard layout usind for experiments in chapter 6. The layout is a slight modification of the Kleine Binckhorst, which is a real shunting yard operated by NS in Den Haag, the Netherlands. The entry track is not represented, but is connected on the bottom left. Next to each track is shown the amount of trains of 80m long fit on it.

In the problem instances used the trains arrive on the entry track located bottom left in the figure. From the entry track they can move onto any of the tracks 52 to 59. All trains have to be serviced at either track 61 or track 62, both of which can be reached directly via one of tracks 56 to 59.

The time it takes to reach either of the service tracks from the entry tracks is the same regardless which of these tracks you move onto first. In fact these tracks are interchangeable in any path if we assume the shunting yard is empty because all of them are connected to the same tracks on either side. We refer to groups of tracks for which this is true as being in the same zone. The yard used for our experiments contains 3 such zones, one containing tracks 53-55, another one containing tracks 56-58 and one containing both service tracks 61 and 62.
However, these symmetries are not guaranteed to hold for the whole duration of the search. If for example track 56 is occupied the choice matters from the perspective of a train on the entry track. Moving onto the occupied track 56 has a different impact on the overall state than moving onto either of the unoccupied tracks 57 or 58.

Not considering these symmetries during search can negatively impact a planner's performance. Such planners will logically evaluate moving onto symmetrical tracks with the same score once it encounters a 'fork in the road'. If one of these symmetrical states is explored further and the planner is not able to select states or which the evaluation is lower than it was at the 'fork in the road' it is wasting time.
We consider one such example where we look at how a julia planner A.2.5 which implements $A*$ search and the h_{add} heuristic [8]. We consider a problem instance for the shunting yard described in figure 6.10 where three trains, $slt40$, $slt41$ and $slt42$ arrive at the entry track in that order.
The planner was able to find an optimal plan where all trains take the most efficient route to a service track and their respective parking tracks. During search, the planner performed **41 expansions** to find the following plan:

```
1: move(train_slt40, track_entry --> track_56)
2: move(train_slt41, track_entry --> track_59)
3: move(train_slt42, track_entry --> track_58)
4: move(train_slt41, track_59 --> track_61_service)
5: service(train_slt41, track_61_service)
6: move(train_slt41, track_61_service --> track_59)
7: move(train_slt40, track_56 --> track_62_service)
8: service(train_slt40, track_62_service)
9: move(train_slt40, track_62_service --> track_56)
10: move(train_slt42, track_58 --> track_61_service)
11: service(train_slt42, track_61_service)
12: move(train_slt42, track_61_service --> track_63)
13: park_different_tracks(train_slt41, train_slt42, track_59, track_63)
14: park_different_tracks(train_slt40, train_slt41, track_56, track_59)
```

There are two moments which can be considered 'forks in the road', the initial state and the state that is reached after action 3. We examine both to understand how the track symmetries are handled by the planner.
For the initial state, visually represented in figure 6.11, the evaluations can be seen in table 6.6. Based on these evaluations there are 4 equally good actions to choose from, out of which moving $train\_slt40$ onto $track\_56$ is chosen. After performing this action the planner was always able to find a path evaluated more favourably than this state was and thus this choice was never reconsidered.
The opposite was the case after the third action in the plan. A visual representation of the resulting state is shown in figure 6.12 and the evaluations calculated for the resulting state are presented in table 6.7.
We can see that there are two cases in which symmetries could be exploited but are not, which leads to inefficiency. As discussed before, tracks 61 and 62 are in the same zone and choosing one or the other to move onto in this case does not make a difference. The planner decides first to explore the state that arises after $train\_slt40$ moves onto $track\_61$. Later it regrets this decision and decides to backtrack before choosing to move $train\_slt40$ onto $track\_62$. This leads to a nearly identical situation down the line, one which the planner will again reject in favour of backtracking to the same 'fork in the road' of figure 6.7.
Starting from this state, at least 10 unnecessary expansions are performed solely due to the planner not being able to exploit zonal symmetries of the shunting yard. When extending the h_{add} heuristic to not expand symmetrical states, fewer nodes are expanded during search. A small experiment was performed to confirm this, the results of which are presented in table 6.8

Table 6.6: Heuristic evaluations for every movement possible in the initial state. Only train slt40 is able to move as it is the only train not blocked by another. The heuristic correctly evaluates that taking any of the tracks 56-59 lead to the most efficient path towards either of the service tracks

Action	g	h	f	Expansions after action
move(train_slt40, track_entry –>track_54)	1	21	22	0
move(train_slt40, track_entry –>track_52)	1	21	22	0
move(train_slt40, track_entry –>track_56)	1	18	19	40
move(train_slt40, track_entry –>track_59)	1	18	19	0
move(train_slt40, track_entry –>track_55)	1	21	22	0
move(train_slt40, track_entry –>track_58)	1	18	19	0
move(train_slt40, track_entry –>track_57)	1	18	19	0
move(train_slt40, track_entry –>track_53)	1	21	22	0

Table 6.7: Heuristic evaluations when choosing the fourth action in the plan. Two cases in which symmetrical states are explored leading to 10 unnecessary expansions solely due to not recognizing symmetries in the shunting yard.

Action	g	h	f	Expansions after action
move(train_slt40, track_56 –>track_60)	4	12	16	0
move(train_slt40, track_56 –>track_61_service)	4	10	14	3
move(train_slt40, track_56 –>track_62_service)	4	10	14	3
move(train_slt42, track_58 –>track_60)	4	11	15	0
move(train_slt42, track_58 –>track_61_service)	4	10	14	7
move(train_slt42, track_58 –>track_62_service)	4	10	14	7
move(train_slt41, track_59 –>track_60)	4	13	17	0
move(train_slt41, track_59 –>track_61_service)	4	10	14	11
move(train_slt41, track_59 –>track_62_service)	4	10	14	0

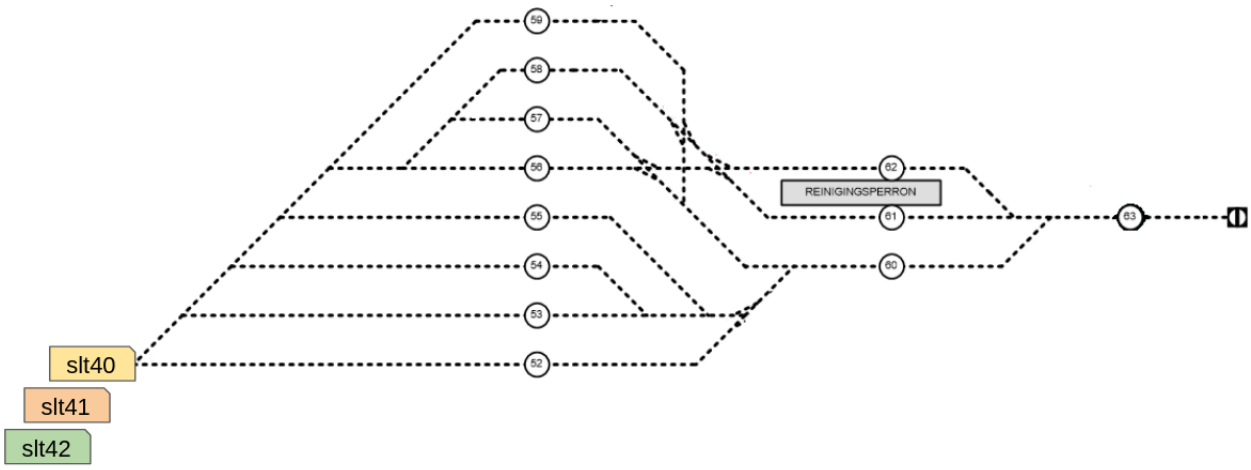

Figure 6.11: Initial state in a simple problem for the modified Kleine Binckhorst layout with 3 trains.

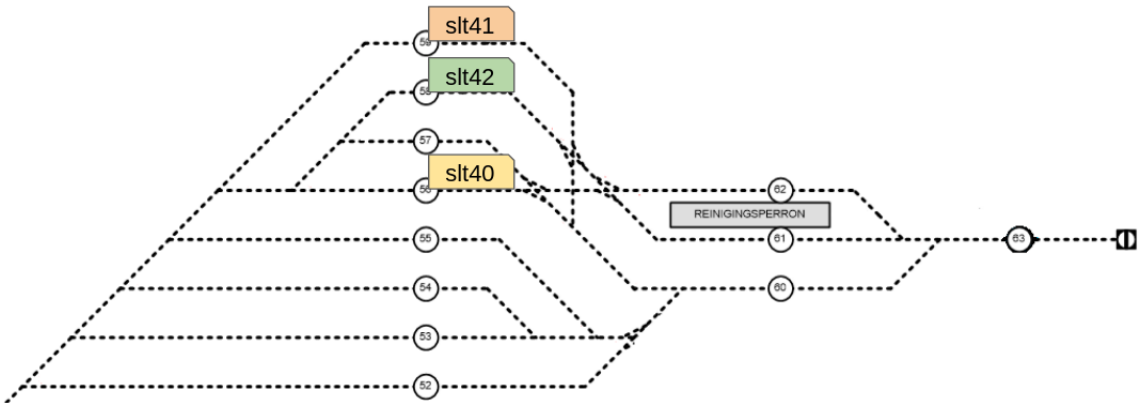

Figure 6.12: State which represents a fork in the road for the planner, for a simple problem for the modified Kleine Binckhorst layout with 3 trains.

Table 6.8: Number of expansions for regular and extended heuristic, showing performance improvement when pruning symmetrical states.

Heuristic	3 trains	4 trains	5 trains
h_{add}	41	41	5642
h_{add} with penalty for symmetrical states	23	31	559

6.3.4. Rush-to-service strategy

Service actions take the longest compared to all other actions in the models. One idea for a strategy that can be implemented by planners is to try and occupy all service tracks as quickly as possible before performing all other necessary movements.

To investigate the potential for this strategy we determine whether a pattern can be observed between the amount of movements performed before all service tracks are occupied for the first time and the makespan of the final solution. In Figure 6.13 we plot the number of initial movements against the makespan of the final plan. In this figure we see a density scatter plot so that we can examine how often a low number of initial movements leads to a shorter makespan.

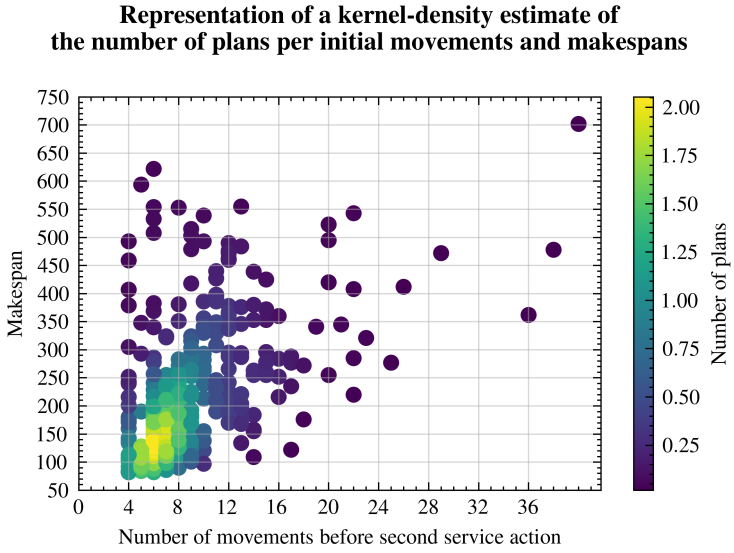

Figure 6.13: Density scatter plot to compare how the initial movements might predict that plan's makespan.

If we follow the dense area we can see an upwards trend. However, there are still plans with relatively short makespans found for a relatively large number of initial movements. What can be inferred is that the shortest makespans are only found for a small amount of initial movements. These results indicate that it is a viable strategy, although not guaranteed to lead to the best performance.

6.3.5. Preferences in parking tracks

In all models parking tracks are decided for each train based on that train's priority and a tracks proximity to the entry track, which we call that track's entry distance. When taking two trains, the one with lower value for its priority needs to be parked either on the same track in front of the other train or on a different track with the same or smaller entry distance than where the higher priority train is parked. In some models the entry distances are made unique. With this a preference is implied regarding the order in which each track will be used for parking. In general indicating a preference beforehand can lead to pruning of the search space and thus might increase efficiency, following a similar concept to the exploitation of symmetries discussed in the previous section.
However, if this preference for parking order is ignored in heuristic evaluations this may lead to inefficiency for the modified Kleine Binckhorst layout specifically. A notable property of this layout is that

the trains need to move over the best parking tracks to reach the service tracks. After the lower priority trains have been serviced they might come to find that the parking track they strongly prefer is already occupied by a higher priority train that still needs to be serviced.

To illustrate this we ran some experiments on a simplified version of the shunting yard used for the experiments in chapter 6, which is shown in figure 6.14. Because of the limited parking space on this small shunting yard it is more likely that trains are inhibiting one another from reaching their goal.
We run a small experiment using a julia planner A.2.5 which implements $A*$ search and the h_{add} heuristic [8]. This planner solves two different problem instances with three trains. In one instance the entry distance of track 57 is smaller than that of track 58, and therefore strongly preferred by lower priority trains whereas in the other instance the entry distances are set to be equal.
The different values for entry distance between the two cause only slight differences in heuristic evaluations. For the problem instance with unique entry distances the lowest priority value train, $train\_slt40$, must be parked on track 57. Because not all trains fit on one single track, both tracks will be used for parking which means that the lowest priority train must be on the track with the lower entry distance between the two.

The results of solving both problem instances are presented in table 6.9. It is clear that with unique entry distances the planner was less efficient. There was one state in particular which caused problems, which is visually represented in figure 6.15.
Up until this point evaluations for each state were exactly the same and thus this state was reached in the same number of expansions. The evaluations in this state for both problem instances are presented in table 6.10. Both planners first choose to move $train\_slt41$, and later reconsider this choice and include the bottom action in their plan instead. However, the precomputed heuristic evaluation for this action is higher for the problem instance with unique entry distances, which means that the planner takes more expansions before reconsidering taking this decision.
Interesting to note is that choosing the action move(train_slt41, track_62 –>track_57) can lead to an optimal solution if both tracks 57 and 58 are equally preferred as a parking track. Ironically, indicating a strong preference for track 57 leads to precomputations that cause problems only if a strong preference for track 57 exists.

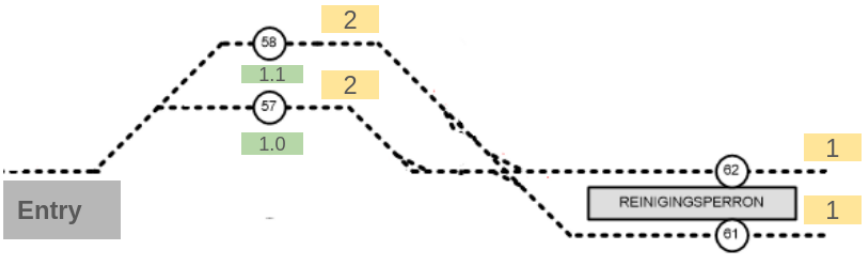

Figure 6.14: Simplified version of the Kleine Binckhorst shunting yard layout which uses only a subset of the original tracks. Because of the few parking tracks conflicts arise more easily.

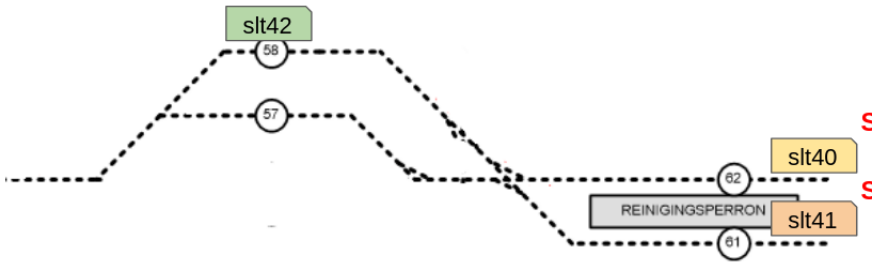

Figure 6.15: State in which the planner starts to move down a dead end. Trains slt40 and slt41 have been serviced, slt42 still needs to visit either of the service tracks.

Table 6.9: Results for solving two problem instances with 3 trains on the simplified Kleine Binckhorst shunting yard. The parking configuration was the same for both but the planner was able to arrive to the solution sooner in the problem.

Problem instance	Expansions	Parking track: train_slt40	Parking track: train_slt41	Parking track: train_slt42
equal entry dist.	21	track_57	track_57	track_58
unique entry dist.	54	track_57	track_57	track_58

Table 6.10: Evaluations for the same state for different problem instances. For both instances the planner chooses to move train_slt41, but in the equal entry distance problem the planner reconsiders this choice sooner and is able to waste less time.

Action	Evaluation equal ed.	Expansions equal ed.	Evaluation unique ed.	Expansions unique ed.
move(train_slt41, track_62 –>track_58)	58	0	34	1
move(train_slt40, track_61 –>track_58)	34	0	54	0
move(train_slt41, track_62 –>track_57)	30	2	30	36
move(train_slt40, track_61 –>track_57)	30	5	34	5

6.4. Conclusions

We set up experiments with several planners and many TUSS models written in PDDL to determine the best performing planner-model pair, which simplifications improved performance and any areas of improvement for the analysed planners.

First we compared the performance of all planner-model pairs. The best performing planner is Temporal FastDownward (TFD), using either Model 3 or Model 16 from category 3. Both models are temporal and model exact train locations. Model 3 only contains movement and service actions. Model 16 does contain driver objects and actions modelling the turning of trains. For both models TFD was able to solve problem instances with up to thirteen trains. The next best performing planner was ENHSP, all other planners performed significantly worse.

The impact of several simplifications were examined for results obtained by the two best performing planners. The best temporal models coupled with TFD consistently outperformed the best results obtained with numeric planners and models. TFD was able to perform better using models without switch actions and where the exact locations of trains were kept track of. Defining strict preferences for parking locations had a significant negative impact on performance. For the other simplifications there is no concrete evidence whether they impact the performance positively or negatively.
The second-best planner, ENHSP, was able to perform best for models with exact train locations, including turns and without driver objects. Similar to TFD, the performance significantly worsened when switch actions were included or strict parking preferences were defined.

Lastly, we examined the contents of the plans created by planners to identify any areas of improvement. We found that the planners we compared are not able to correctly identify and prune symmetrical states. Neither symmetries in state values nor in the shunting yard layout were handled optimally. Furthermore, we identified that planners which more evenly distributed service tasks across the available service tracks were able to produce plans with shorter makespans. In the plans obtained using our experiments it was more likely to find shorter makespans if fewer movement actions were performed before all service tracks were occupied. Lastly we identified that strict preferences for parking tracks is more likely to negatively influence performance when planners are not able to anticipate on this when looking far ahead from a state.

7

Train Order Preserving Search

When examining the contents of the plans obtained by experimentation we found how several aspects of the TUSS models were not handled optimally by the model-independent planners. However, many properties of the PDDL models and planners were convenient, such as the filtering of unavailable actions from each state and how the planning problem is organised in a search tree. In this section, we explore how we can improve an existing planner, leveraging the beneficial features but modifying the search algorithm and heuristic evaluation to handle TUSS-specific complexities better. As a result we present a TUSS-model-dependent PDDL planner called Train Order Preserving Search (TOPS). This planner prioritises actions that help either attain or maintain the departure order of trains included in the problem instance.
First, we motivate our choice of planner to be modified and explain what properties are kept the same. Next, we explain our modifications to the search algorithm and heuristic evaluation. Finally, we compare how TOPS performs compared to the best planners found after experimental analysis in Chapter 6.

7.1. Planner to be modified

Because of time constraints we choose to modify the SymbolicPlanners [56] package written in julia [6]. Although the performance with this package on the TUSS models was poor compared to others, it is most easily modified since the code for heuristic evaluations for all other planners was provided as a compiled package. The code for SymbolicPlanners is open-source and available on Github and is more easily modified.
This package includes either a forward or a backwards search function and several different heuristics. We choose to modify the forward search function, which is an implementation of weighted A^* search because this is most commonly found in other PDDL planners. This is combined with a custom developed heuristic.
The planner selected for modification is a numeric planner and thus only is able to handle numeric models.

7.2. Search algorithm

As mentioned before, the TOPS planner uses a modified version of the A^* search algorithm. Several modifications are introduced to improve search-space pruning, inspired by the experimental results from model-independent planners. We add mechanisms to recognise and prune symmetrical states.
In sections 6.3.2 and 6.3.3 we identified that other planners did not prune states that were already encountered and did not recognise which tracks in a shunting yard lead to similar paths. The modifications introduced in TOPS will ensure that these symmetries are pruned correctly.
Moreover, we introduce the following three rules:

1. When a service action is available it is the only action considered
2. If not all trains can be parked, no park action is considered

3. If all trains can be parked, it is only allowed to execute park actions

The motivation for each modification is discussed followed by the pseudo code of the resulting search algorithm.

7.2.1. Pruning symmetrical states

In this sub-section we explain how we accomplish the pruning of symmetrical states. In the domains the $(total - cost)$ fluent is both the metric to be minimised and included in the state values.
During an expansion in our search algorithm the first step is to remove this fluent from the state and it is stored separately. Nodes in the search tree consist of the state values without the cost fluent, the value for the cost and the action sequence that lead to the state. This allows us to to compare two nodes based on the states relevant to the configuration on the shunting yard and focus the search on exploring new states.

7.2.2. Recognising zonal symmetries

In Section 6.3.3 we discussed what we call zonal symmetries and how they are not recognised by the additive heuristic in the SymbolicPlanners package. To summarize, when two different tracks can fit the same number of trains and are connected to the same set of tracks on either side, we consider them to be in the same zone. When both tracks are empty, it does not matter which of the two tracks a train moves onto.

Before search starts the shunting yard layout is analysed to determine which zone each track is in. During expansion of a node the available grounded actions for the state in that node are identified. In the action parameters the name of any unoccupied track is replaced by the name of the zone it is in. Duplicate grounded actions are ignored for the next expansion.

7.2.3. Parking and service rules

Service actions are made mandatory because rarely if ever a situation occurs where it is beneficial for an unserviced train to wait on a service track. The planner can never backtrack to a state in which a train has just moved onto a service track, which means that a small part of the search space is pruned.

Only collective parking is allowed out of concern that the planner needs to backtrack far to reconsider the decision to park a train. This modification is included to simulate the functionality of derived predicates, where in each state it is verified whether all sub-goals have been achieved. Once all trains can be parked only park actions are allowed to focus on producing a plan so that at least some solution can be presented before search continues.

7.2.4. Pseudo-code

The pseudo-code for this search algorithm is presented below in algorithm 2.

Algorithm 2 Modified A^* search algorithm used in TOPS

1: given $s_{initial}, T$
2: Initialise $OPEN$
3: $OPEN \leftarrow (s_{initial}, 0, 0, \{\})$
4: **while** $OPEN$ not empty **do**
5: $n_{current} \leftarrow$ node with lowest priority in $OPEN$
6: **if** any service action in $available\_actions(n_{current})$ **then**
7: $filtered\_available\_actions \leftarrow \{first\_service\_action(available\_actions(n_{current}))\}$
8: **else**
9: **if** all trains can be parked **then**
10: $filtered\_available\_actions \leftarrow \{first\_park\_action(available\_actions(n_{current}))\}$
11: **else**
12: $filtered\_available\_actions \leftarrow \{\}$
13: **for** act_i in $available\_actions(n_{current})$ **do**
14: **if** act_i is movement $\&$ $destination\_track(act_i)$ is empty **then**
15: replace $name_{destination\_track(act_i)}$ with $name_{zone(destination\_track(act_i)}$
16: **end if**
17: **if** $act_i \notin filtered\_available\_actions$ **then**
18: $filtered\_available\_actions \cup \{act_i\}$
19: **end if**
20: **end for**
21: **end if**
22: **end if**
23: **for** act_i in $filtered\_available\_actions$ **do**
24: $s_i \leftarrow act_i(state(n_{current}))$
25: $priority \leftarrow f(s_i)$
26: $cost \leftarrow cost(n_{current}) + cost(act_i)$
27: $path \leftarrow path(n_{current}) \cup \{act_i\}$
28: $n_i \leftarrow (s_i, priority, cost, path)$
29: **if** s_i in T **then**
30: return n_i
31: **end if**
32: **if** $\exists n_j \in OPEN$ for which $state(n_j) \equiv s_i$ **then**
33: **if** $cost(n_j) \leq cost$ **then**
34: next iter
35: **end if**
36: **if** $cost(n_j) > cost$ **then**
37: $OPEN \leftarrow OPEN \setminus \{n_j\}$
38: **end if**
39: **else**
40: $OPEN \cup \{n_i\}$
41: **end if**
42: **end for**
43: $OPEN \leftarrow OPEN \setminus \{n_{current}\}$
44: **end while**

If we more closely examine the steps in this algorithm:

- **Step 1:** In the problem definition we are provided with the inital state $s_{initial}$ and the set of goal nodes T.
- **Step 6-8:** If any service action is found it is mandatory to execute it.
- **Step 9-11:** If all trains can be parked no other type of action is considered.
- **Step 12-21:** Remove any actions for zonally-symmetric states.
- **Step 24-28:** The node contents are slightly different compared to regular A^* search, we include

the cost separately from the state.

- **Step 29-31:** Once we have reached a goal state we return the solution. In its original form A^* search terminates when the first solution is found although in this version the algorithm is extended such that search continues.
- **Step 32-39:** If we have encountered another node wherein the same state is stored, we keep the node with lowest cost. This is how symmetrical states are pruned.
- **Step 39-41:** If we encounter a new state we add the node to the open list.

7.3. Order Preserving Heuristic

In this section we discuss the heuristic evaluation used in the TOPS planner. With this heuristic we leverage contextual knowledge about TUSS problem instances to guide the search. The departure order in a problem instance will be used to assign priorities, where the train that is set to depart first is assigned the lowest value for priority. Based on these the heuristic will prefer moving trains with lowest priority values towards their destination.
Furthermore we can easily determine the shortest paths to either service tracks or the entry track. This can be used to determine the direction in which every train prefers to move. This in turn can be used to identify states in which the preferred direction clashes for different trains, which is not at all what we want.
Lastly, we analysed the plans obtained using model-independent planners and the insights from section 6.3.4 indicate that a rush-to-service strategy might lead to plans with shorter makespans.
The heuristic evaluation is divided into multiple penalty terms. The contextual knowledge we have about TUSS problems informs us how to determine these. In the sub-sections we first discuss what pre-computations are performed to be able to calculate the $weighted\_total\_distance$, $ordering$, and $service\_capacity$ penalty terms. Next, we discuss the $entry\_track\_occupation$ penalty, which is only preferred in specific problem instances. All penalty terms are multiplied by weights, the values of which were determined in preliminary experiments. Finally, we briefly discuss whether this heuristic evaluation is admissible or not.

7.3.1. Pre-computations

To calculate some penalty terms we need to obtain certain knowledge about the TUSS problem instance. We recall that the input consists of a description of a shunting yard layout and an arrival schedule. The output contains a departure schedule and a list of required service tasks to be done for each train-unit to be shunted. The input is formulated as an initial state, and the output is formulated as a set of goal-states.
First we create a graph which represents a relaxation of the provided PDDL problem instance. In this graph, nodes represent a track and arcs are drawn between tracks that are linked in the shunting yard layout. All arcs have the same weight.
The goal state in the provided problem instance describes one or several desired orders in which trains should exit the shunting yard. These configurations are used to infer a priority for each train. If one train is always supposed to depart the yard earlier than another it will have a lower priority value assigned.
Moreover, the following properties are calculated for each track using the relaxed graph representation of the shunting yard:

- The direction and distance to the closest service track
- The direction and distance to the entry track

Since this relaxation of the problem is often very easy to solve one can use any shortest-path algorithm to extract the relevant information. We opt for Dijkstra's algorithm [16].
These properties are used to calculate the penalty terms. If a train moves to a state for which the distance to its destination is further away the heuristic will evaluate this state transition unfavourably.

7.3.2. Weighted total distance penalty

The idea behind the Weighted Total Distance penalty (WTD) is to penalise trains for moving away from their sub-goals. Penalties are harsher for trains with lower priority values. Branches in the search tree where trains increase their distance to their sub-goals should be considered only if there is no

alternative, as it is very likely to require drivers to enter- and exit trains more often which increases the cost and leads to a sub-optimal plan.
This penalty term makes use of the pre-computed distance values in the relaxed problem. For each train we look up the the distance from its current location to the location of its next sub-goal. The sum of distances of all trains make up the total distance penalties:

$$WTD = w_d \frac{2}{N} \sum_{t}^{N} (N+1-priority_t) * [x_t * 2 * (dist_{service}(loc_t) + dist_{entry}(loc_s)) + (1 - x_t) * dist_{entry}(loc_t)] \tag{7.1}$$

$$x_t = \begin{cases} 1 \text{ if train } t \text{ is supposed to be serviced but has not been serviced yet} \\ 0 \text{ otherwise} \end{cases} \tag{7.2}$$

$$loc_t = \text{ current location of train } t, \tag{7.3}$$

$$loc_s = \text{ location of the service track which is closest to train } t \tag{7.4}$$

This penalty term increases any time a train moves farther away from its destination, which can either be a service track or the entry track. Trains with lower priority values are penalised more harshly for this than the others.

Preliminary experiments showed that penalising distances of un-serviced trains harsher compared to serviced trains lead to better performance. This modification causes the planner to employ the rush-to-service strategy where the first trains arriving on the entry track are moved to service tracks before other actions are considered.

7.3.3. Blockage penalty

In the desired ordering in the goal state, no trains with higher priority values are uninhibited by those with lower priority values. The BP term evaluates orderings that resemble the desired final ordering more favourably. The aim is to explore areas of the state space where the lowest priority trains are able to move most freely.
To calculate the BP we make use of the pre-computed preferred directions of each train in the relaxation of the provided TUSS instance. For any train that cannot move in its preferred direction because it is blocked by another train this penalty term is incremented. Train A is blocked by train B if either of the following is true:

- A and B need to move in opposite directions but cannot do so without colliding
- A and B need to move in the same direction but the train with the higher priority value is in front of the other train

$$BP = \sum_{t_i} \sum_{t_j \neq t_i} w_b * b_{ij} * p_{ij} \tag{7.5}$$

$$b_{ij} = \begin{cases} 1 \text{ if train } t_j \text{ is blocking } t_i \text{ in its preferred direction} \\ 0 \text{ otherwise} \end{cases} \tag{7.6}$$

$$p_{ij} = \begin{cases} 1 \text{ if train } t_j\text{'s priority value is higher than that of } t_i \\ 0 \text{ otherwise} \end{cases} \tag{7.7}$$

7.3.4. Ordering penalty

Another approach to steer search towards ensuring the desired final ordering is done in the OP term. In a state we want to incentivize the lowest priority trains to be closer to their sub-goal. However we want to dis-incentivize the higher priority trains to move further away from their sub-goals to ensure this.
To calculate the OP term we look up the distances of all trains to their sub-goals. These distances are sorted by their values and then stored in the $expected\_distance$ array. In another array the distances are sorted by the priority value of the corresponding train in the $actual\_distance$ array. If the $actual\_distance$ is larger than the $expected\_distance$, the OP is increased.

$$OP = \sum_{t} max\left[(exp\_dist(t) - act\_dist(t)), 0\right] \tag{7.8}$$

7.3.5. Entry track occupation penalty

Whether or not to include this penalty depends on the input provided for a problem instance. Since we modify a numeric planner there is no way to indicate the arrival times of trains in a schedule. If the individual arrival times of incoming trains is spaced apart significantly, it makes more sense to perform all required actions for a train before moving another onto the entry track.
However, in the opposite case where trains arrive practically at the same time, it is preferred to move all trains onto the shunting yard to free up space for the next trains arriving at the entry track. In these types of problem instances it would be beneficial to include the Entry Track Occupation (ETO) penalty term

$$ETO = w_e to \sum_{t}^{N} x_t \tag{7.9}$$

$$x_t = \begin{cases} 1 \text{ if train } t \text{ is on the entry track} \\ 0 \text{ otherwise} \end{cases} \tag{7.10}$$

7.3.6. Admissibility

We recall the definition of an admissible heuristic to be one that never overestimates the actual cost of an optimal plan.
The cost of TUSS plans are dominated by the amount of times a driver enters and exits a train, whereas the heuristic value of OPH is determined by the train distances and amount of displaced trains. In practice it becomes evident that the properties used to calculate OPH are indeed relevant to the total cost of the resulting plan, as the performance of search is drastically improved over other heuristics. However it is impossible to provide mathematical proof that the heuristic value will never exceed the actual cost of an optimal plan, making the OPH heuristic inadmissible.

7.4. Analysing performance

Now that we have explained how the TOPS planner searches for plans it is time to compare its performance to the best model-independent planner. First we explain the setup of experiments with which we obtain results for the TOPS planner. Then we look at the results and compare performances of TOPS and Temporal FastDownward on TUSS problems modelled in PDDL.

7.4.1. Experimental setup

Since the TOPS planner is an extension of the SymbolicPlanners package, it can only handle numeric domains. Plans are produced for one simple and one more detailed model. The simple model only contains movement and service actions, but the costs for turns are included in the effects of the movement actions. The more detailed model has the same actions as the simple one, but also contains action definitions describing walking to, entering, exiting and turning trains. Both models contain exact train locations.

In Section 6.3.1 it was shown that planners which evenly distributed service tasks performed better than planners which did not. Both models force the TOPS planner to alternate the location of service tasks.

We use the same method of generating problem instances as in Section 6.1. Again we use the modified layout of the Kleine Binckhorst shunting yard, the same scenario where trains arrive and have to depart in the same order and all trains have to be serviced before being parked. The same time limits are set for the TOPS planner as well as the post-processing routine

7.5. Results

To generate results we used two different configurations of TOPS, one including ETO penalty and one excluding it. Both configurations were used to solve both the simple and detailed variations of the problem instances described in the experimental setup in this chapter.

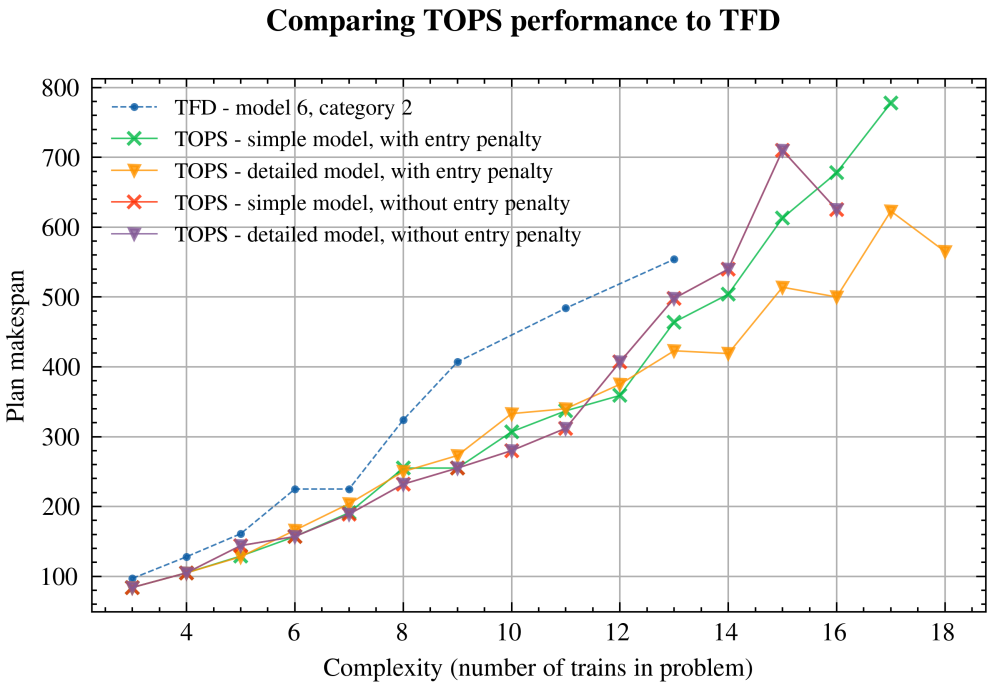

Figure 7.1: Comparison of performance four configurations of TOPS and the best model-independent planner-model pair.

In Figure 7.1 we can clearly see that both regarding the makespans and the maximum solvable complexity all configurations of TOPS significantly outperform the best planner-model pair found in Chapter 6.

8

Conclusion and Further Research

In this paper, we set out to explore how PDDL can be used to solve TUSS problem instances. First, we examined how the matching, parking, routing, and servicing sub-problems could be integrated into a detailed model. In this detailed model, we also investigated the inclusion of actions performed by train drivers. We found that PDDL provides the necessary tools to describe solutions to all sub-problems and driver actions. However, although possible, the features are not well-suited to model the matching sub-problem, and when this is included in the detailed model, it negatively impacts both readability and state size.
In addition to the description of a detailed model, we provided numerous ways in which TUSS aspects could be simplified. When combined with a post-processing routine, including optimisation using a CP solver, these simplified models produce plans with a similar level of detail to the detailed PDDL model.
Furthermore, we analysed how several domain-independent PDDL planners performed when solving TUSS problem instances modelled in PDDL. After analysing performances for multiple planner-model pairs, we identified that Temporal FastDownward was the best-performing planner. It consistently produced plans with shorter makespans and handled the most complex instances containing up to thirteen trains. The next-best-performing planner was ENHSP, which was able to solve instances with up to eleven trains. The worst-performing planners were OPTIC and the SymbolicPlanners package, which were slightly outperformed by Metric-FF and Numeric FastDownward.
Performances per simplification were also analysed. We found that modelling exact train locations, lenient parking preferences, and excluding switch actions significantly improves performance for most planners.
The contents of plans obtained by our experiments were analysed to identify potential improvements that could be applied in our new planner called Train Order Preserving Search (TOPS). We identified that the model-independent planners did not handle symmetries optimally. It was worth attempting to implement a strategy where the trains rush to fill service track capacity initially, and service tasks are distributed as evenly as possible. These improvements were implemented in TOPS, combined with a custom heuristic evaluation that aims to either obtain or preserve the order of trains in the departure schedule.
This resulted in a TUSS-specific PDDL planner that significantly outperformed the best model-independent planner. TOPS was able to solve instances with up to eighteen trains while producing plans with significantly shorter makespans.

Due to time constraints, the search algorithm for one of the worst-performing planners was examined because it was the easiest to modify. Examining the code of some of the better-performing planners might provide other valuable insights into improving the performance of planners for the TUSS models. This same planner was used as a starting point for the TOPS planner, also for convenience. It might be interesting to explore implementing some ideas in one of the better-performing planners. The preprocessing method for these planners might be one of the reasons for their better performance, and if so, it would be a good idea to leverage that.
The solving method in TOPS is very promising as it is able to find high-quality plans for complex in-

stances with very few expansions. However, we cannot compare its performance fairly to the current state-of-the-art solving methodology. This is because the problem instances used for experimentation in this thesis differ significantly and are less realistic. This might be an interesting idea to explore in further research.

References

[1] Constructions Aeronautiques et al. “Pddl| the planning domain definition language”. In: *Technical Report, Tech. Rep.* (1998).

[2] Kamenga et al. “Train Unit Shunting : Integrating rolling stock maintenance and capacity management in passenger railway station”. In: (2019).

[3] Johannes Aldinger and Bernhard Nebel. “Interval based relaxation heuristics for numeric planning with action costs”. In: *Joint German/Austrian Conference on Artificial Intelligence (Künstliche Intelligenz)*. Springer. 2017, pp. 15–28.

[4] C. Athmer. “An Evolutionary Algorithm for the Train Unit Shunting and Servicing Problem”. In: (2021).

[5] J Benton, Amanda Coles, and Andrew Coles. “Temporal planning with preferences and time-dependent continuous costs”. In: *Proceedings of the International Conference on Automated Planning and Scheduling*. Vol. 22. 2012, pp. 2–10.

[6] Jeff Bezanson et al. “Julia: A fast dynamic language for technical computing”. In: *arXiv preprint arXiv:1209.5145* (2012).

[7] Avrim L. Blum and John C. Langford. “Probabilistic Planning in the Graphplan Framework”. In: *Recent Advances in AI Planning*. Ed. by Susanne Biundo and Maria Fox. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 319–332. ISBN: 978-3-540-44657-6.

[8] Blai Bonet and Héctor Geffner. “Planning as heuristic search”. In: *Artificial Intelligence* 129.1 (2001), pp. 5–33. ISSN: 0004-3702. DOI: https://doi.org/10.1016/S0004-3702(01)00108-4. URL: https://www.sciencedirect.com/science/article/pii/S0004370201001084.

[9] Nils Boysen et al. “Shunting yard operations: Theoretical aspects and applications”. In: *European Journal of Operational Research* 220.1 (2012), pp. 1–14. ISSN: 0377-2217. DOI: https://doi.org/10.1016/j.ejor.2012.01.043. URL: https://www.sciencedirect.com/science/article/pii/S0377221712000811.

[10] Roel van den Broek, Han Hoogeveen, and Marjan van den Akker. “How to measure the robustness of shunting plans”. In: *18th workshop on algorithmic approaches for transportation modelling, optimization, and systems (atmos 2018)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik. 2018.

[11] Roel van den Broek, Han Hoogeveen, and Marjan van den Akker. “Personnel scheduling on railway yards”. In: *20th Symposium on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2020)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik. 2020.

[12] Geoffrey Chu. *the chuffed solver*. https://github.com/chuffed/chuffed. 2023.

[13] Amanda Coles et al. “Forward-chaining partial-order planning”. In: *Proceedings of the International Conference on Automated Planning and Scheduling*. Vol. 20. 2010, pp. 42–49.

[14] D.A. Van Cuilenborg. “Train Unit Shunting Problem, a Multi-Agent Pathfinding approach”. In: (2020).

[15] Gabriele Di Stefano and Magnus Love Koči. “A Graph Theoretical Approach To The Shunting Problem”. In: *Electronic Notes in Theoretical Computer Science* 92 (2004). Proceedings of ATMOS Workshop 2003, pp. 16–33. ISSN: 1571-0661. DOI: https://doi.org/10.1016/j.entcs.2003.12.020. URL: https://www.sciencedirect.com/science/article/pii/S1571066104000052.

[16] E. W. Dijkstra. “A note on two problems in connection with graphs”. In: *Numerische Mathematik* 1 (1959), pp. 269–271.

[17] Vladimir Estivill-Castro and Jonathan Ferrer-Mestres. “Path-finding in dynamic environments with PDDL-planners”. In: *2013 16th International Conference on Advanced Robotics (ICAR)*. IEEE. 2013, pp. 1–7.

[18] Patrick Eyerich, Robert Mattmüller, and Gabriele Röger. “Using the Context-enhanced Additive Heuristic for Temporal and Numeric Planning”. In: *ICAPS*. 2009.

[19] Richard Freling et al. “Shunting of passenger train units in a railway station”. In: *Transportation Science* 39.2 (2005), pp. 261–272.

[20] Giorgio Gallo and Federico Di Miele. “Dispatching buses in parking depots”. In: *Transportation Science* 35.3 (2001), pp. 322–330.

[21] Malik Ghallab. *Automated Planning: Theory and Practice*. Morgan Kaufmann, 2004.

[22] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated planning and acting*. Cambridge University Press, 2016.

[23] Jørgen Thorlund Haahr, Richard M. Lusby, and Joris Camiel Wagenaar. “Optimization methods for the Train Unit Shunting Problem”. In: *European Journal of Operational Research* 262.3 (2017), pp. 981–995. ISSN: 0377-2217. DOI: https://doi.org/10.1016/j.ejor.2017.03.068. URL: https://www.sciencedirect.com/science/article/pii/S0377221717302990.

[24] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. “A Formal Basis for the Heuristic Determination of Minimum Cost Paths”. In: *IEEE Transactions on Systems Science and Cybernetics* 4.2 (1968), pp. 100–107. DOI: 10.1109/TSSC.1968.300136.

[25] Malte Helmert and Héctor Geffner. “Unifying the Causal Graph and Additive Heuristics.” In: *ICAPS*. 2008, pp. 140–147.

[26] Malte Helmert, Gabriele Röger, et al. “How Good is Almost Perfect?.” In: *AAAI*. Vol. 8. 2008, pp. 944–949.

[27] Jörg Hoffmann. “A heuristic for domain independent planning and its use in an enforced hill-climbing algorithm”. In: *International Symposium on Methodologies for Intelligent Systems*. Springer. 2000, pp. 216–227.

[28] Jörg Hoffmann. “The Metric-FF Planning System: Translating“Ignoring Delete Lists”to Numeric State Variables”. In: *Journal of artificial intelligence research* 20 (2003), pp. 291–341.

[29] P.M. Jacobsen. “Train shunting at a workshop area”. MA thesis. 2011.

[30] Wenrong Jiang. “Analysis of iterative deepening a* algorithm”. In: *IOP Conference Series: Earth and Environmental Science*. Vol. 693. 1. IOP Publishing. 2021, p. 012028.

[31] Leo G Kroon, Ramon M Lentink, and Alexander Schrijver. “Shunting of passenger train units: an integrated approach”. In: *Transportation Science* 42.4 (2008), pp. 436–449.

[32] Leo G Kroon, Ramon M Lentink, and Alexander Schrijver. “Shunting of passenger train units: an integrated approach”. In: *Transportation Science* 42.4 (2008), pp. 436–449.

[33] Ryo Kuroiwa, Alexander Shleyfman, and J Christopher Beck. “LM-Cut Heuristics for Optimal Linear Numeric Planning.” In: *Proc. ICAPS*. 2022, pp. 203–212.

[34] Ramon Lentink. *Algorithmic decision support for shunt planning*. 73. 2006.

[35] Ramon M Lentink et al. “Applying operations research techniques to planning of train shunting”. In: *Planning in Intelligent Systems: Aspects, Motivations, and Methods* (2006), pp. 415–436.

[36] Nazariy Lonyuk. *TUSP PDDL models*. https://github.com/LonyuNaz/tusp-pddl-post-processing. 2024.

[37] Nazariy Lonyuk. *TUSP PDDL models*. https://github.com/LonyuNaz/tusp-pddl-models. 2024.

[38] Hang Ma. “Graph-based multi-robot path finding and planning”. In: *Current Robotics Reports* 3.3 (2022), pp. 77–84.

[39] Drew M McDermott. “The 1998 AI planning systems competition”. In: *AI magazine* 21.2 (2000), pp. 35–35.

[40] Jesse Mulderij et al. “Train unit shunting and servicing: a real-life application of multi-agent path finding”. In: *arXiv preprint arXiv:2006.10422* (2020).

[41] neighthan. *tfd*. https://github.com/neighthan/tfd. 2023.

[42] Luuk van Nes. “Planning Drivers for Shunting Yards”. MA thesis. 2024.

[43] Nicholas Nethercote et al. “MiniZinc: Towards a standard CP modelling language”. In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2007, pp. 529–543.

[44] Francesca Rossi, Peter Van Beek, and Toby Walsh. “Constraint programming”. In: *Foundations of Artificial Intelligence* 3 (2008), pp. 181–211.

[45] Enrico Scala, Patrik Haslum, and Sylvie Thiébaux. “Heuristics for numeric planning via subgoaling”. In: *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*. IJCAI’16. New York, New York, USA: AAAI Press, 2016, pp. 3228–3234. ISBN: 9781577357704.

[46] Enrico Scala et al. “Interval-based relaxation for general numeric planning”. In: *ECAI 2016*. IOS Press, 2016, pp. 655–663.

[47] Enrico Scala et al. “Search-guidance mechanisms for numeric planning through subgoaling relaxation”. In: *Proceedings of the International Conference on Automated Planning and Scheduling*. Vol. 30. 2020, pp. 226–234.

[48] Alexander Shleyfman, Ryo Kuroiwa, and J Christopher Beck. “Symmetry Detection and Breaking in Cost-Optimal Numeric Planning.” In: *Proc. ICAPS*. 2023, pp. 393–401.

[49] Alexander Shleyfman, Ryo Kuroiwa, and J Christopher Beck. “Symmetry Detection and Breaking in Cost-Optimal Numeric Planning.” In: *Proc. ICAPS*. 2023, pp. 393–401.

[50] Nederlandse Spoorwegen. *NS Jaarverslag 2023*. 2024. URL: https://www.nsjaarverslag.nl/ (visited on 05/08/2024).

[51] Kees Szabó. “Scheduling mechanics on a Shunting Yard: skills, synchronization and train movements”. MA thesis. 2023.

[52] Ayal Taitler et al. *The 2023 International Planning Competition*. 2024.

[53] A. Tate. In: *Encyclopedia of the Cognitive Sciences* (1999).

[54] Roel van den Broek et al. “A Local Search Algorithm for Train Unit Shunting with Service Scheduling”. English. In: *Transportation Science* 56.1 (2022). Green Open Access added to TU Delft Institutional Repository ’You share, we take care!’ - Taverne project https://www.openaccess.nl/en/you-share-we-take-care Otherwise as indicated in the copyright section: the publisher is the copyright holder of this work and the author uses the Dutch legislation to make this work public., pp. 141–161. ISSN: 0041-1655. DOI: 10.1287/trsc.2021.1090.

[55] Roel W Van Den Broek. “Train Shunting and Service Scheduling: an integrated local search approach”. MA thesis. 2016.

[56] ztangent. *SymbolicPlanners.jl*. 2024. URL: https://juliaplanners.github.io/SymbolicPlanners.jl/dev/.

A

Additional information regarding experiments

A.1. Models

Below we provide a table per model category that contain all models of that category used in experiments. We recall that models of category 1 and 2 include concurrent actions whereas category 3 and 4 do not. Models in category 1 and 3 describe train locations exactly whereas categories 2 and 4 use relative locations.

Table A.1: Contents of all models in category 1

Model	Domain	Turns	Drivers	Switch actions	Walk actions	Strict parking preferences	Duplicate sub-goals
1	1a	no	no	no	no	no	no
2	1a	no	no	no	no	no	yes
3	1b	no	no	no	no	no	no
4	1c	no	no	no	no	yes	no
5	1d	no	no	yes	no	no	no
6	2a	yes	no	no	no	no	no
7	2b	yes	no	no	no	no	no
8	2c	yes	no	no	no	no	no
9	3a	no	yes	no	no	no	no
10	3b	no	yes	no	no	no	no
11	3b	no	yes	no	no	no	yes
12	3c	no	yes	no	no	no	no
13	3d	no	yes	no	no	yes	no
14	3e	no	yes	yes	no	no	no
15	4a	yes	yes	no	no	no	yes
16	4b	yes	yes	no	no	no	no
17	5a	no	yes	no	yes	no	no
18	5b	no	yes	no	yes	no	no
19	6a	yes	yes	no	yes	no	no

Table A.2: Contents of all models in category 2

Model	Domain	Turns	Drivers	Switch actions	Walk actions	Strict parking preferences	Duplicate sub-goals
1	1a	no	no	no	no	no	no
2	1a	no	no	no	no	no	yes
3	1b	no	no	no	no	no	no
4	1c	no	no	no	no	yes	no
5	2a	yes	no	no	no	no	no
6	2b	yes	no	no	no	no	no
7	2b	yes	no	no	no	no	yes
8	3a	no	yes	no	no	no	no
9	3b	no	yes	no	no	no	no
10	3c	no	yes	no	no	no	no
11	4a	yes	yes	no	no	no	no
12	5a	no	yes	no	yes	no	no
13	5b	no	yes	no	yes	no	no
14	6a	yes	yes	no	yes	no	no

Table A.3: Contents of all models in category 3

Model	Domain	Turns	Drivers	Switch actions	Walk actions	Strict parking preferences	Duplicate sub-goals
1	1a	no	no	no	no	no	no
2	1b	no	no	no	no	yes	no
3	1c	no	no	no	no	no	no
4	2a	yes	no	no	no	no	no
5	2a	yes	no	no	no	no	yes
6	2b	yes	no	no	no	no	no
7	2b	yes	no	no	no	no	yes
8	2c	yes	no	no	no	yes	no
9	3a	no	yes	no	no	no	no
10	3b	no	yes	no	no	no	no
11	3c	no	yes	no	no	no	no
12	1d	no	no	no	no	no	no
13	2d	yes	no	no	no	no	no
14	3d	no	yes	no	no	no	no
15	4a	yes	yes	no	no	no	no
16	4b	yes	yes	no	no	no	no
17	6a	yes	yes	no	yes	no	no
18	6b	yes	yes	no	yes	no	no
19	1e	no	no	no	no	no	no

Table A.4: Contents of all models in category 4

Model	Domain	Turns	Drivers	Switch actions	Walk actions	Strict parking preferences	Duplicate sub-goals
1	1a	no	no	no	no	no	no
2	1a	no	no	no	no	no	yes
3	1b	no	no	no	no	yes	no
4	1c	no	no	yes	no	no	no
5	1d	yes	no	no	no	no	no
6	2a	yes	yes	no	no	no	no
7	2b	yes	yes	no	no	no	no
8	2c	no	yes	no	no	yes	no
9	3a	no	no	no	no	no	no
10	3b	no	no	no	no	no	no
11	3c	yes	no	no	no	no	no
12	3d	no	no	no	no	no	yes
13	4a	no	yes	no	no	no	no
14	5a	yes	yes	no	yes	no	no

A.2. Planners

In this section we provide a short description of the planners compared in the experiments.

A.2.1. Temporal Fast Downward

The first version of Temporal Fast Downward (TFD) [18] participated in the temporal satisfycing track of the 6th IPC in 2008. It uses the context-enhanced additive heuristic [25] which solves a hierarchy of local solutions in order to estimate distances to goal states. In our experiments we use a modified version of TFD v0.4 [41] which was entered into the IPC in 2014. This version supports domains that are numeric and/or temporal.

A.2.2. OPTIC

The temporal planner called Optimizing Preferences and TIme-dependent Costs (OPTIC), developed by Benton et al. [5] offers a new approach in which reasoning is done with plan quality metrics not directly correlated to the total duration of a plan. This is relevant for domains in which plan quality depends on the time each individual sub-goal is achieved rather than the total duration. The planner builds upon techniques introduced in POPF, developed by Coles et al [13]. OPTIC introduces dummy plan steps and MIP encodings of preferences combined with a preference-aware heuristic to compute plans for numeric and/or temporal domains.

A.2.3. ENHSP

The Expressive Numeric Heuristic Search Planner (ENHSP) was developed by Scala et al [45]. The planner supports many features, but preliminary results show best performance using weighted A^* search guided by the MRP extraction heuristic [47]. This heuristic notes how many times certain actions need to be executed before a goal state is reached in a relaxation of the problem. This notation is then used to better define sub-goals and helpful actions.

A.2.4. Metric-FF

Metric-FastForward (Metric-FF) is a numeric planner that uses a search function guided by a heuristic function. The values of this heuristic are calculated by solving a relaxed version of the problem with Graphplan [7] algorithm. The original problem is relaxed by modifying the effects of every action such that once a value for a literal has been achieved it is never unset.

A.2.5. SymbolicPlanners

Developed in the programming language Julia [6], the SymbolicPlanners package [56] supports both the forward and backwards A^* search algorithm combined with the additive and maximizing heuristics,

much like what is possible in the ENHSP planner but with different pre-processing. With this package it is possible to compute solutions for numeric domains.

A.2.6. Numeric FastDownward

Numeric FastDownward (NFD) planner was the best performing in the numeric track of the most recent IPC in 2023 [52]. It was originally developed by Aldinger et al in 2017 [3]. The most recent improvements were made in symmetry breaking by Shleyman et al [49] and the linear numeric landmark-cut heuristic by Kuroiwa et al [33]. Preliminary experiments showed that performance on TUSS models was best using the symmetry-breaking DKS_A^* search algorithm with the interval-relaxation-based additive heuristic.

A.3. All experimental results

Below we present all results obtained for every planner-model pair analysed. Results for SymbolicPlanners and OPTIC are omitted as they were never able to solve instances with more than 3 trains. The same holds for Metric-FF and Numeric FastDownward and all models in category 4.

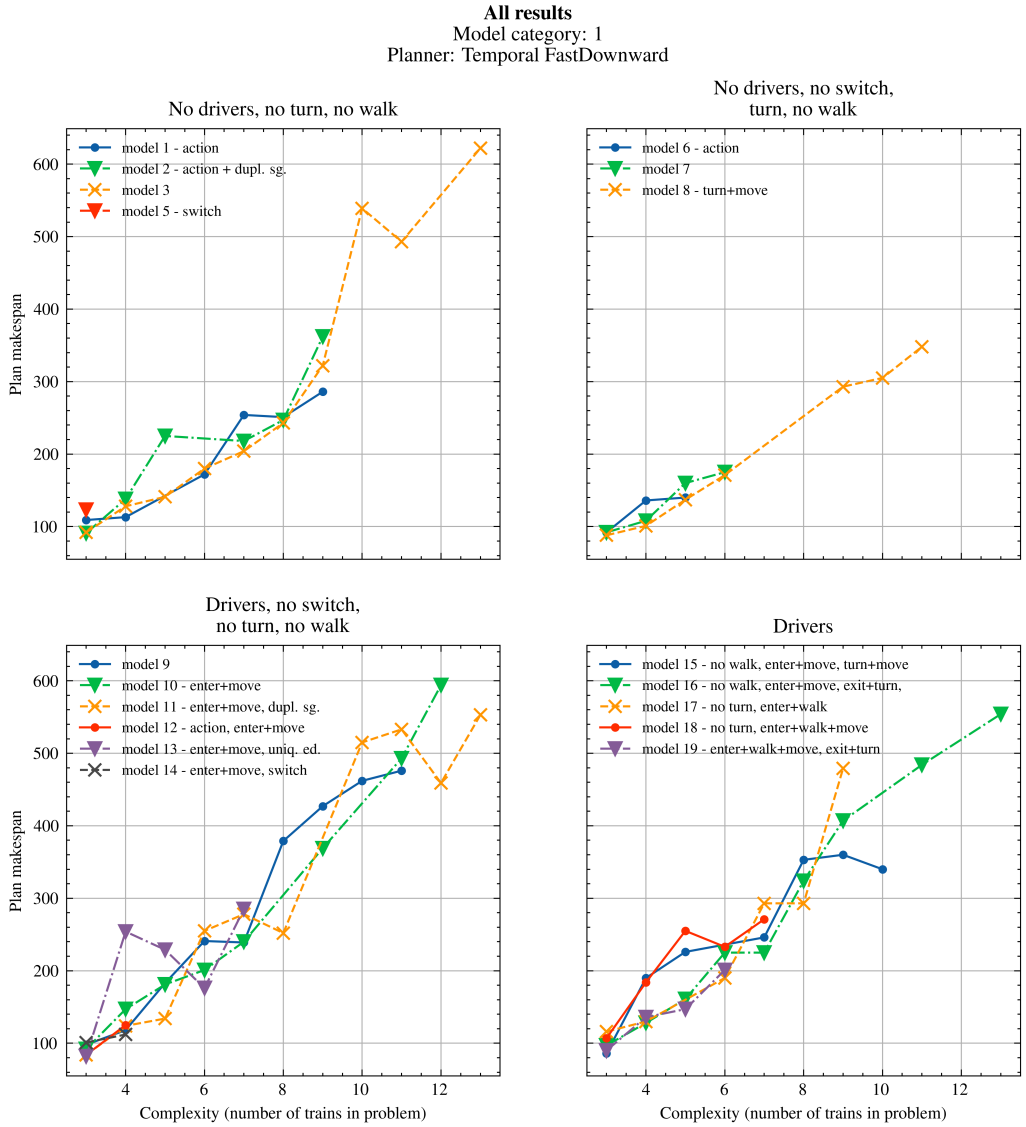

Figure A.1: All results found for planner TFD and model category 1

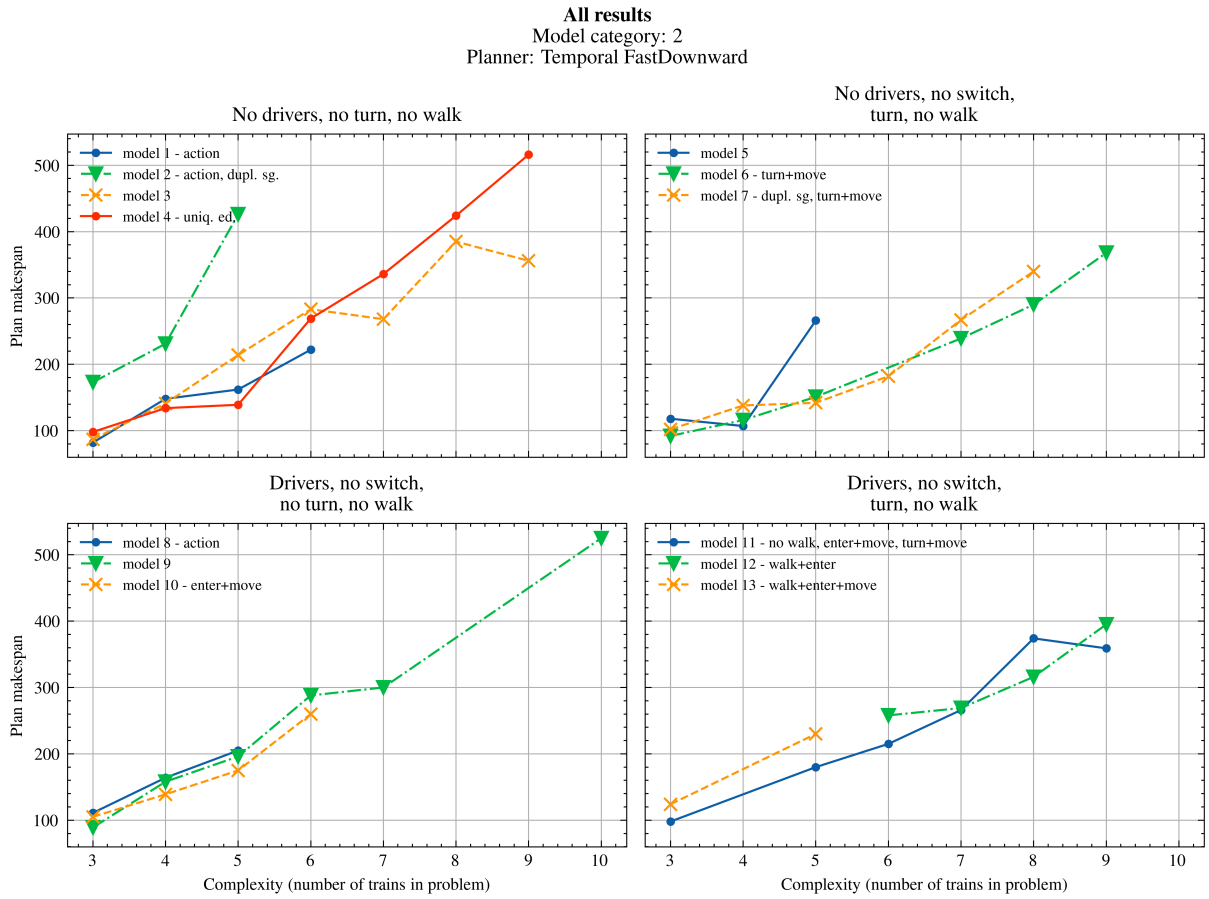

Figure A.2: All results found for planner TFD and model category 2

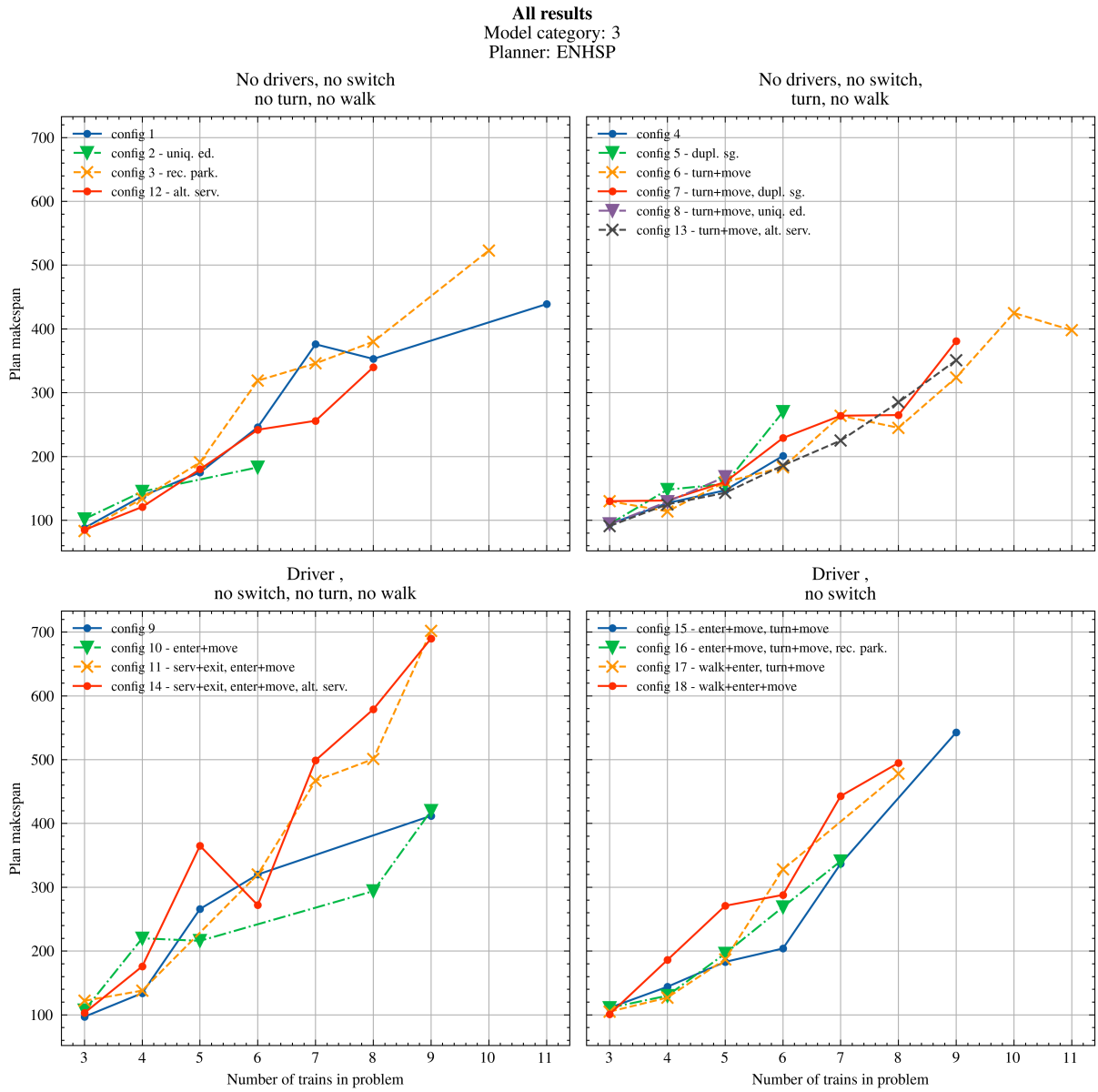

Figure A.3: All results found for planner ENHSP and model category 3

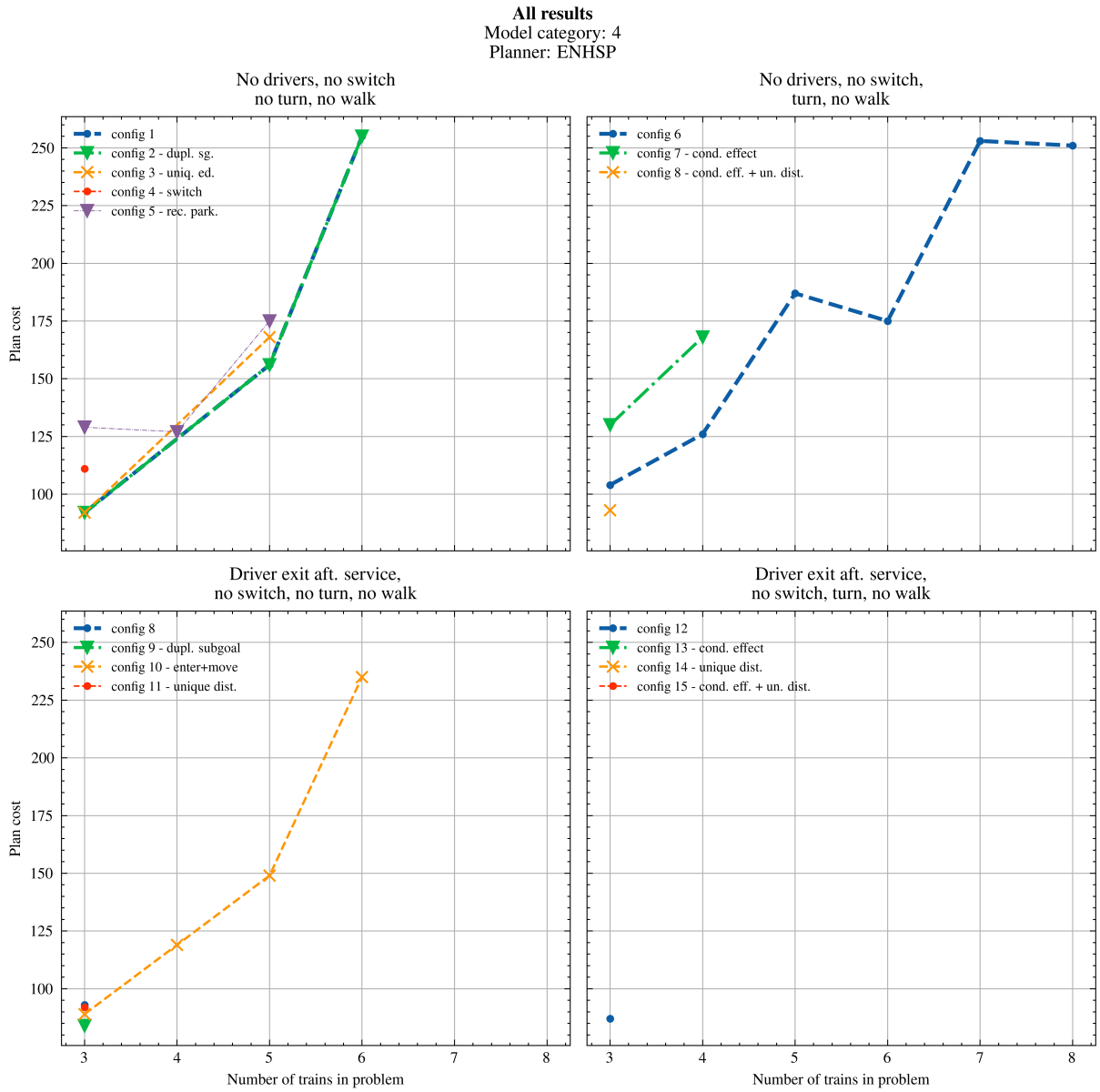

Figure A.4: All results found for planner ENHSP and model category 4

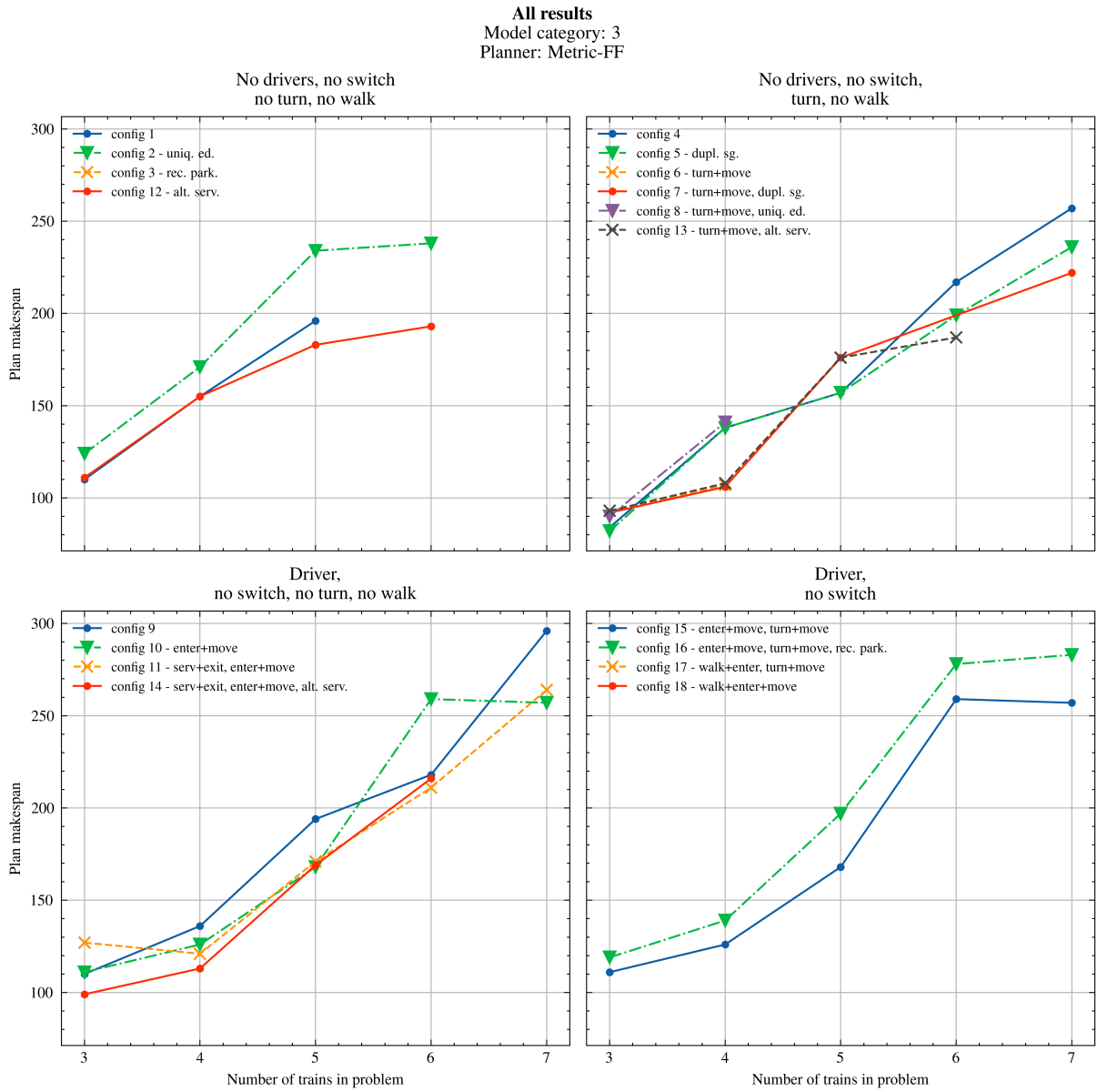

Figure A.5: All results found for planner Metric-FF and model category 3

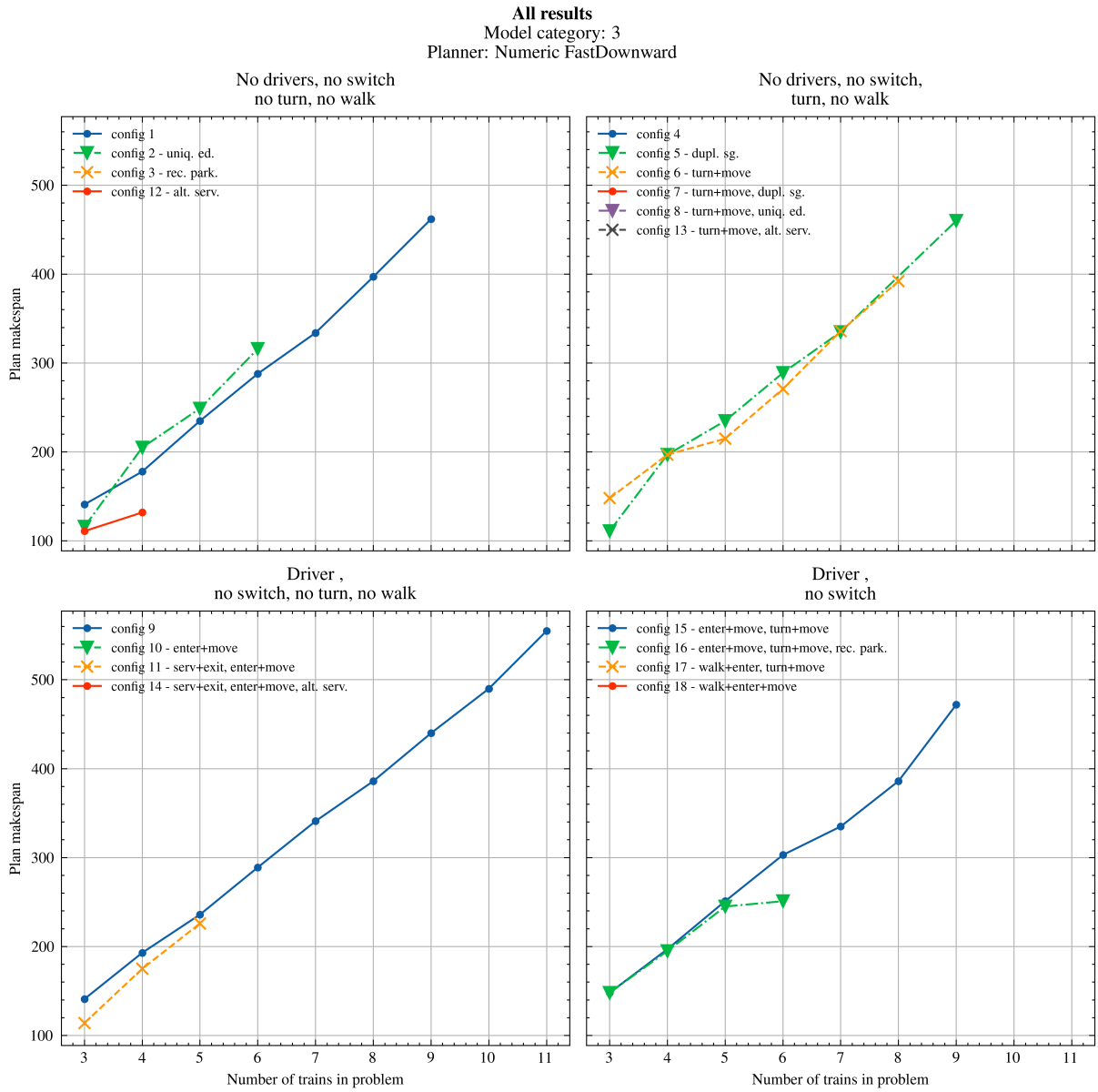

Figure A.6: All results found for planner NFD and model category 3